MySQL Basics and Tools

Percona Training

http://www.percona.com/training

InnoDB Basics and Tools

STORAGE ENGINES

Storage Engines

- MySQL Separates SQL *from* Storage.
  - Replication, Partitioning, Stored Procedures all happen above the storage engine layer.
  - Storage happens in the Storage Engines.
- The most popular storage engine is InnoDB.
  - The default is InnoDB as of MySQL 5.5.
  - For 99% of people MyISAM is probably *the wrong choice*.

Storage Engines: InnoDB

- Most popular, default since MySQL 5.5.
- Row-level locking.
- ACID transactions.
- Automatic crash recovery.
- Caching data and indexes.
- Referential integrity (foreign keys).
- Better performance and scalability when tuned well.
- Fulltext indexing in MySQL 5.6.

Storage Engines: MyISAM

- Default storage engine until MySQL 5.1.
- Table-level locking.
- Relies on filesystem caching—risk of corruption.
- Fulltext indexing.
- GIS indexing.

Storage Engines: Others

- MEMORY
  – Stores data in volatile memory.
- BLACKHOLE
  – Stores no data, like /dev/null. Very fast! Small footprint!
  – Useful as a dummy target, while DML is written to the binlog.
- CSV
  – Stores data in text files using a comma-separated value format.
- ARCHIVE
  – Store large amounts of unindexed data with transparent compression.
  – Supports only INSERT and SELECT.

Storage Engines: Not Recommended

- MERGE
  - Interface to a collection of identical MyISAM tables as one table.
  - Use Partitioning instead.
- FEDERATED
  - Lets you access data from remote MySQL instances without using replication or cluster technology.
  - Roughly analogous to Oracle Database Links.
  - Not recommended; stability and performance issues.

Changing a Table's Storage Engine

- Simple to convert:

```
mysql> ALTER TABLE name ENGINE=InnoDB;
```

- It performs a *table restructure* (just like many ALTER statements do), and the table is locked for the duration.
- Test carefully—you could truncate data or lose table details if data types or index types are not supported in the new storage engine.

The MySQL Server

- Start & stop MySQL Server with the init script:
  `$ /etc/init.d/mysql [start|stop|restart|status]`

- Some Linux distributions also support this style:
  `$ service mysql [start|stop|restart|status]`

- The init script launches `mysqld_safe`. This watchdog script runs the daemon `mysqld`, and restarts the daemon if it exits abnormally.

The MySQL Server (cont.)

- The `mysqld` daemon runs many threads for all the work of listening for client connections, running queries, logging, doing I/O, etc.

```
$ pstree -a
init
  ├─mysqld_safe /usr/bin/mysqld_safe --datadir=/var/lib/mysql
--pid-file=/var/run/mysqld/mysqld.pid
  |   └─mysqld --basedir=/usr --datadir=/var/lib/mysql --
plugin-dir=/usr/lib64/mysql/plugin --user=mysql--pid-file=/v
  |       ├─{mysqld}
  |       ├─{mysqld}
  |       ├─{mysqld}
```

The MySQL Client

- The `mysql` client runs SQL commands interactively, or executes an SQL script in batch mode.
- You can enable default client options in $HOME/.my.cnf:

```
[client]
host = db1
user = scott
password = tiger
```

Client Builtins

- `mysql> pager` *`command`*
  - Filter output through a shell program. Turn off with `nopager`.

- `mysql> tee` *`file`*
  - Log session to a file. Turn off with `notee`.

- `mysql> warnings`
  - Show any warnings by default. Turn of with `nowarning`.

Client Builtins

- `mysql> edit`
  - Edit the current command in $EDITOR.
- `mysql> prompt` *`string`*
  - Add metacharacters to prompt.
- `mysql> delimiter` *`string`*
  - Use *string* as statement terminator instead of default `;`
  - Needed for `CREATE TRIGGER` / `PROCEDURE` / `FUNCTION`, because those statements include unquoted `;` characters.

Client Builtins

- Vertical format output: `\G` statement terminator.

Client Builtins

- **editline** for simple command editing.
  - Control-A / E: move cursor to start / end of line.
  - Control-W: erase to start of line.
  - Control-R: search SQL command history.
- View current editline key bindings:
  - Edit $HOME/.editrc and temporarily add this line:

```
bind
```

  - Start mysql client. It outputs all key bindings.

* readline was used in MySQL Community builds prior to 5.6, except on Windows.

PERCONA TRAINING

Other MySQL Tools

- `mysqladmin`: Run administration commands as arguments, making it easier to write scripts.
- `mysqldump`: Logical database dump tool.
- `mysqlbinlog`: Convert binary logs to SQL scripts.
- `mysqlimport`: Bulk load flat files to database.

MySQL GUI Tools

- MySQL Workbench
  - http://dev.mysql.com/downloads/workbench/
  - Browse database objects.
  - Prototype and test SQL queries.
  - Edit data model diagrams.
  - Administer server instances.

- MySQL Enterprise Monitor
  - Commercial tool available to subscribers of Oracle Support.
  - Monitoring and alerting for one or many MySQL instances.
  - Advisors for tuning and fixing issues.

Many Third-Party GUI Tools

dbForge Studio	http://www.devart.com/dbforge/mysql/studio/	Free to $99
HeidiSQL	http://www.heidisql.com/	Free (GPL)
Navicat	http://www.navicat.com/	Free to $369
phpMyAdmin	http://www.phpmyadmin.net/	Free (GPL)
Sequel Pro	http://www.sequelpro.com/	Free (GPL)
SQLYog / MonYog	http://www.webyog.com/	$99+ / $199+
Toad for MySQL	http://www.quest.com/toad-for-mysql/	Free

Note: Command Line Only

- Today's examples assume use of the MySQL command line.
  - If you prefer to use MySQL Workbench or another GUI environment, you may do so on your own.

InnoDB Basics and Tools

STORAGE ENGINES

Storage Engines

- MySQL Separates SQL *from* Storage.
  – Replication, Partitioning, Stored Procedures all happen above the storage engine layer.
  – Storage happens in the Storage Engines.
- The most popular storage engine is InnoDB.
  – The default is InnoDB as of MySQL 5.5.
  – For 99% of people MyISAM is probably *the wrong choice*.

Storage Engines: InnoDB

- Most popular, default since MySQL 5.5.
- Row-level locking.
- ACID transactions.
- Automatic crash recovery.
- Caching data and indexes.
- Referential integrity (foreign keys).
- Better performance and scalability when tuned well.
- Fulltext indexing in MySQL 5.6.

Storage Engines: MyISAM

- Default storage engine until MySQL 5.1.
- Table-level locking.
- Relies on filesystem caching—risk of corruption.
- Fulltext indexing.
- GIS indexing.

Storage Engines: Others

- MEMORY
  - Stores data in volatile memory.
- BLACKHOLE
  - Stores no data, like /dev/null. Very fast! Small footprint!
  - Useful as a dummy target, while DML is written to the binlog.
- CSV
  - Stores data in text files using a comma-separated value format.
- ARCHIVE
  - Store large amounts of unindexed data with transparent compression.
  - Supports only INSERT and SELECT.

Storage Engines: Not Recommended

- MERGE
  - Interface to a collection of identical MyISAM tables as one table.
  - Use Partitioning instead.
- FEDERATED
  - Lets you access data from remote MySQL instances without using replication or cluster technology.
  - Roughly analogous to Oracle Database Links.
  - Not recommended; stability and performance issues.

Changing a Table's Storage Engine

- Simple to convert:

```
mysql> ALTER TABLE name ENGINE=InnoDB;
```

- It performs a *table restructure* (just like many ALTER statements do), and the table is locked for the duration.
- Test carefully—you could truncate data or lose table details if data types or index types are not supported in the new storage engine.

Replication

Percona Training

http://www.percona.com/training

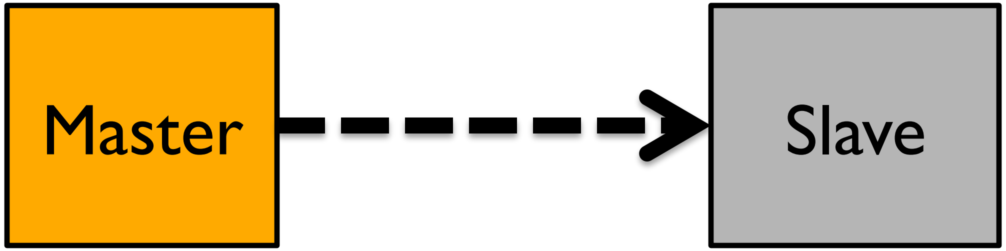
Table of Contents

Replication

OVERVIEW

Replication Overview

- Replication is a mechanism for recording a series of changes on one database server and applying the same changes to a replica.
- The source the “master” and its replica is a “slave.”

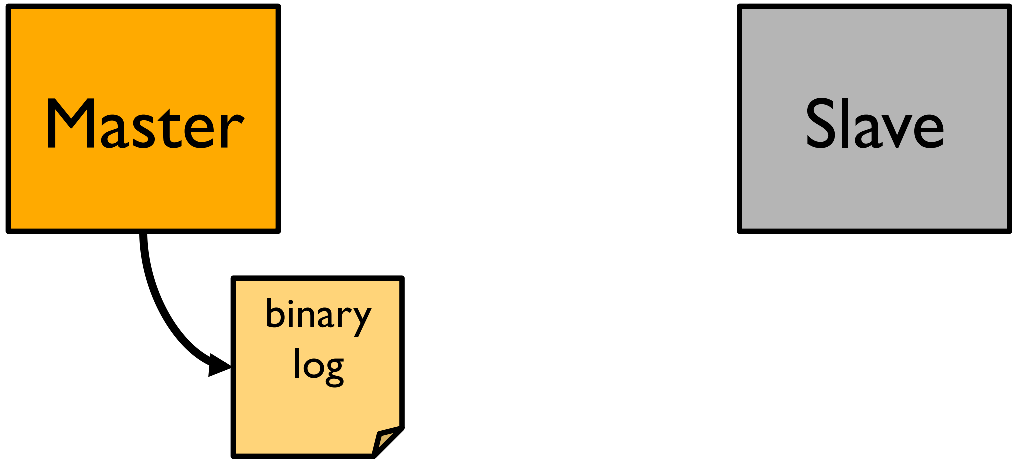


Replication Solutions

High Availability	If the master server crashes, the slave serves a hot spare.
Load Balancing	The application can send some read queries to the slave, giving you greater capacity for read-only query load.
Backups	You can create database backups on a slave, without worrying about impacting production traffic.
Dedicated Queries	Reports or other offline tasks can read data from a slave.
Data Distribution	The slave can be an off-site replica that is continually up to date.
Testing	Experiment with queries, MySQL tuning, or version upgrades you aren't ready to use on the master.

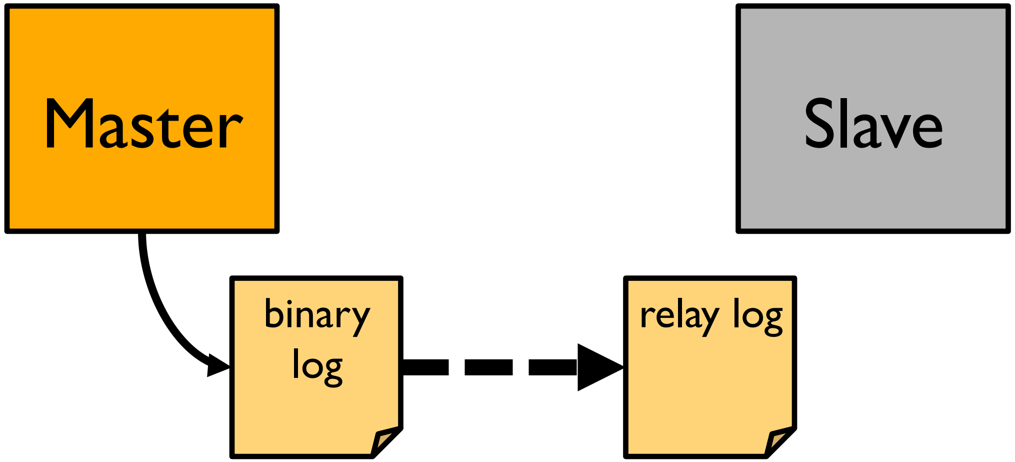
How Replication Works

- Master records committed changes in its *binary log*.



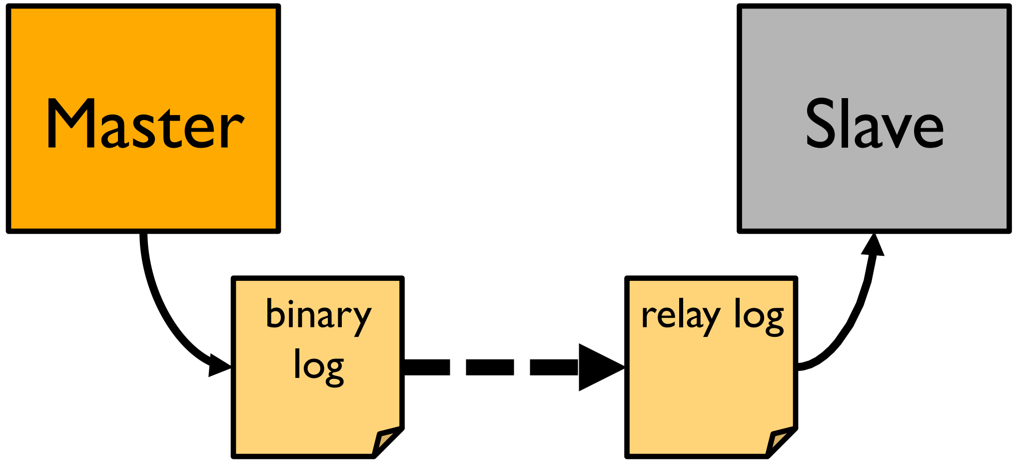
How Replication Works

- The slave's IO thread continually downloads the master's binary logs.
- These copies on the slave are called *relay logs*.



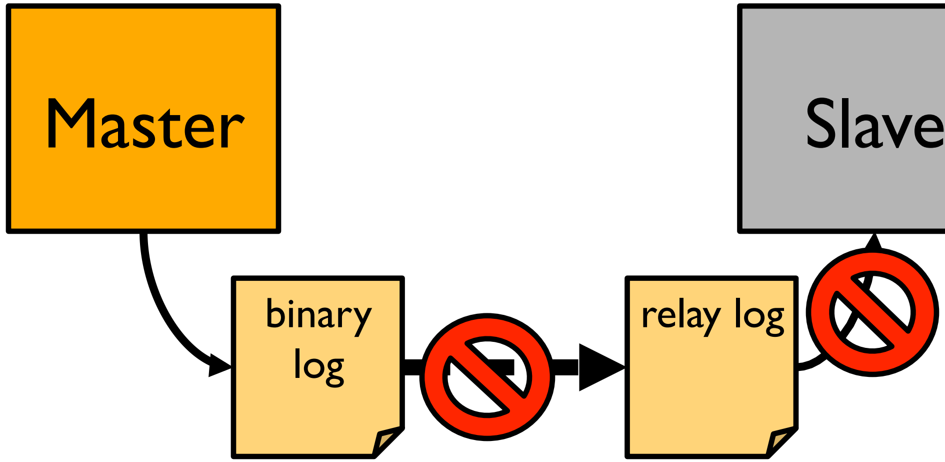
How Replication Works

- The slave's replication SQL thread executes the changes against its copy of the database.
- They stay in sync as incremental changes are applied.



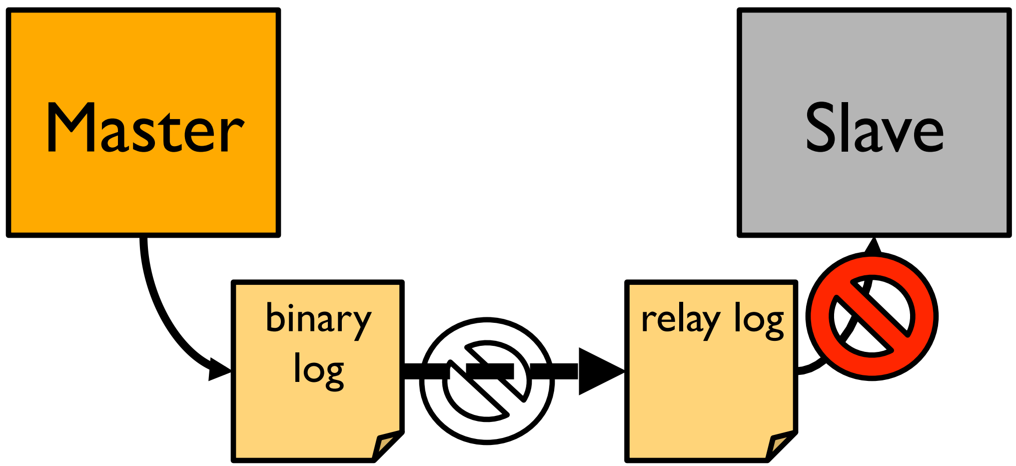
How Replication Works

- Replication is asynchronous by default.
- The slave can stop executing changes or stop downloading logs, and resume later where it left off.



Semi-Synchronous Replication

- Commit on master waits for at least one semi-sync slave to confirm receipt of the binary log.
- Assures a change is logged in two places—although slave can still lag behind executing the changes.



Clarification on the Binary Log

- In Oracle and some other RDBMS implementations, the transaction log is also used for replication.
- MySQL has two separate change logs.
  - InnoDB transaction log: physical changes to InnoDB data pages, to ensure durability. Used only during crash recovery.
  - Binary log: representing logical changes to data. These logs are used for replication and point-in-time recovery.

Replication

SETTING UP

Setting Up Replication

1. Enable binary logs on the master.
2. Assign a server id to each server.
3. Grant a user on the master server.
4. Initialize the slave with a replica of data.
5. Configure the slave.
6. Start replication.

1. Enabling Binary Logs

- The log-bin config variable names a filename prefix for binlog files.
- MySQL will generate a numeric suffix, with incrementing values as it allocates new files.
- Configure in `/etc/my.cnf`:

```
[server]
log_bin
```

- Enabling/disabling the binary log or changing the file prefix requires restart of the MySQL instance.

2. Assigning Server-Id

- Each MySQL instance in a replication chain must have a distinct server id.
- Any distinct positive integer between 1 and 2^{32}-1.
- Server id 0 means the instance cannot be a master or a slave.
- Configure in `/etc/my.cnf`:

```
[server]
server_id = 1234
```

- Changing the server id requires restart of the MySQL instance.

3. Creating the Replication User

- The slave needs to connect to the master to download binary logs.
- The user must minimally have REPLICATION SLAVE privilege.
- You may also grant REPLICATION CLIENT privilege so this user can run commands to report replication status.

```
mysql> GRANT REPLICATION SLAVE,
REPLICATION CLIENT ON *.* TO
repl@'192.168.0.%' IDENTIFIED BY 'xyzzy';
```

4. Initialize Data for the Slave

- Changes in the binary log are incremental, so the master and slave should start with a common baseline of data.
- The most important thing to do is note the current binary log file and position when you capture the initial data.
- For example, you can include the binary log coordinates in any backup from the master.

```
$ mysqldump --master-data=1 …other options…
```

- Restore the data dump on the slave.

Locating Master Binlog Coordinates

- You can also view the current binlog position on the master at the time you capture the initial data you use for the slave:

```
mysql> SHOW MASTER STATUS\G
            File: mysql-bin.000023
        Position: 107
```

5. Configure Replication

- Run on the slave:

```
mysql> CHANGE MASTER TO
         MASTER_HOST='masterdb',
         MASTER_USER='repl',
         MASTER_PASSWORD='xyzzy',
         MASTER_LOG_FILE='mysql-bin.000023',
         MASTER_LOG_POS=107;
```

- Use the log file and pos you noted on the master when you created the initial data.

5. Configure Replication

- Older versions of MySQL supported options in `/etc/my.cnf` to configure the slave, but this is deprecated.
  - Bad idea anyway, since your server may restart, and reset the binlog coordinate the slave subscribes to.

6. Start Replication

- Run on the slave:

```
mysql> START SLAVE;
```

- To stop the slave:

```
mysql> STOP SLAVE;
```

- You can also independently start and stop the IO thread (downloading binary logs) and the SQL thread (executing relay logs):

```
mysql> START SLAVE IO_THREAD;
mysql> START SLAVE SQL_THREAD;
```

Check Replication Status (1)

```
mysql> SHOW SLAVE STATUS\G
                  Master_Host: 192.168.56.110
                  Master_User: repl
                  Master_Port: 3307
             Master_Server_Id: 2
                Connect_Retry: 60
              Master_Log_File: db2.000019
          Read_Master_Log_Pos: 302
        Relay_Master_Log_File: db2.000019
          Exec_Master_Log_Pos: 302
```

Continued...

Check Replication Status (2)

```
   Slave_IO_State: Waiting for master to send event
 Slave_IO_Running: Yes
Slave_SQL_Running: Yes
Seconds_Behind_Master: 0
        Last_Errno: 0
        Last_Error:
     Last_IO_Errno: 0
     Last_IO_Error:
    Last_SQL_Errno: 0
    Last_SQL_Error:
      Skip_Counter: 0
```

Continued…

Check Replication Status (3)

```
  Relay_Log_File: relay.000007
   Relay_Log_Pos: 4
 Relay_Log_Space: 107
 Until_Condition: None
  Until_Log_File:
   Until_Log_Pos: 0
```

Continued…

Check Replication Status (4)

```
          Replicate_Do_DB:
      Replicate_Ignore_DB:
Replicate_Ignore_Server_Ids:
        Replicate_Do_Table:
    Replicate_Ignore_Table:
   Replicate_Wild_Do_Table:
Replicate_Wild_Ignore_Table:
```

Continued…

Check Replication Status (5)

```
         Master_SSL_Allowed: No
         Master_SSL_CA_File:
         Master_SSL_CA_Path:
            Master_SSL_Cert:
          Master_SSL_Cipher:
             Master_SSL_Key:
Master_SSL_Verify_Server_Cert: No
```

http://dev.mysql.com/doc/refman/5.6/en/replication-administration-status.html

Exercise: Set Up Replication

1. Configure replication between two instances.
2. Start replication and check replication status.
3. Verify that replication is running, by creating a dummy table on the master and then look for it on the slave.
```
mysql> CREATE TABLE test.foo
   (id INT PRIMARY KEY);
```
4. Stop replication.
5. Create another dummy table on the master, look for it on the slave. It should not be there yet.
6. Start replication and look for the new table on the slave again.

Replication

UNDER THE HOOD

Replication Under the Hood

- Binary log formats
- More on log files
- Chains of replication
- Replication filtering

Binary Log Formats

- You can set the default binary log format on the master, in `/etc/my.cnf`:

```
[server]
binlog_format = STATEMENT
binlog_format = ROW
binlog_format = MIXED
```

- In theory, you can change this dynamically, but some errors have been reported when attempting this on a busy server.
- To be safe, at least stop applications from making changes before changing `binlog_format` globally.

Statement Based Binary logs

- Binary log can contain SQL statements to be executed.
- Slave re-parses SQL statements from relay log and executes against its replica data.
- Sensitive to discrepancies in data. Applying changes against wrong data can propagate and worsen the drift.
- Some statements are by nature non-deterministic, or have different effects on the master vs. the slave:

```
UPDATE tablename SET ...
WHERE columnname > SYSDATE();
```

Row Based Binary Logs

- Binary log contains the result of changes executed on the master. That is, copies of the rows affected by changes.
- Applying on the slave does not run SQL, it simply replaces the rows.
- Pros:
  - Avoids re-executing costly statements, possibly reducing CPU load on the slave.
  - Protects against slave drift in many cases.
  - Reduces locks necessary to ensure changes are applied in the correct order.

Row Based Binary Logs

- Cons:
  - When a statement applies to many rows, *all* affected rows need to be copied into the binary log file.
  - Binary logs contain the row image *before* and *after* the change.
  - MySQL 5.6 mitigates this, storing only columns that changed.

Mixed Mode Binary Logs

- Defaults to STATEMENT format, and typically uses STATEMENT almost all the time.
- Switches to ROW format only for statements that MySQL detects are unsafe for replication.
- DDL (CREATE/ALTER/DROP) is always logged in STATEMENT format regardless.

Auxiliary Replication Files

mysql-bin.index	MySQL uses this file to catalog its binary logs exist.
mysql-relay-bin.index	A slave catalogs its relay logs.
master.info	A slave stores its replication parameters (that you set with CHANGE MASTER). E.g. the replication password in *plain text*.
relay-log.info	A slave uses this file to record how far it's executed changes. • Percona Server tracks a slave in a crash-safe way: http://www.percona.com/doc/percona-server/5.5/reliability/innodb_recovery_update_relay_log.html • MySQL 5.6 does this too.

Suppressing Binary Logging

- You can make changes in a session without logging.

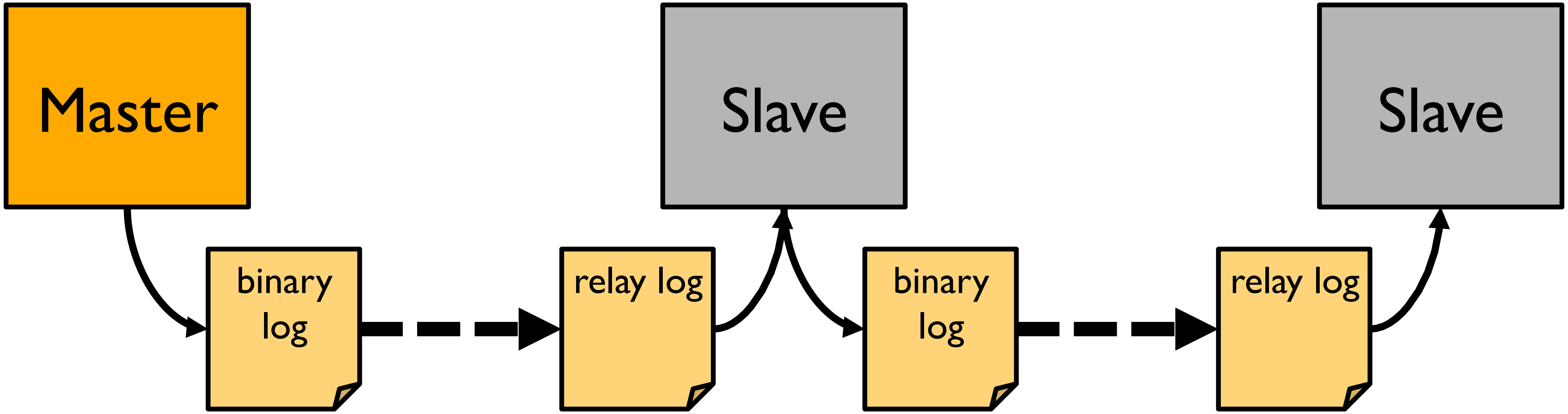
```
mysql> SET SESSION SQL_LOG_BIN=0;
mysql> ALTER TABLE title
    ADD INDEX (title(50), production_year);
```

- Resume logging by setting the variable back to 1, or else simply end the current session.
- Common technique to reduce downtime:
  - Apply changes to a slave.
  - Swap the roles of a slave and its master.
  - Apply changes to the former master.

Chains of Replication

- A slave can be the master of a downstream slave. The middle slave must write to its own binary log.

```
[server]
log_bin = 1
log_slave_updates = 1
```



Chains and Binlog Format

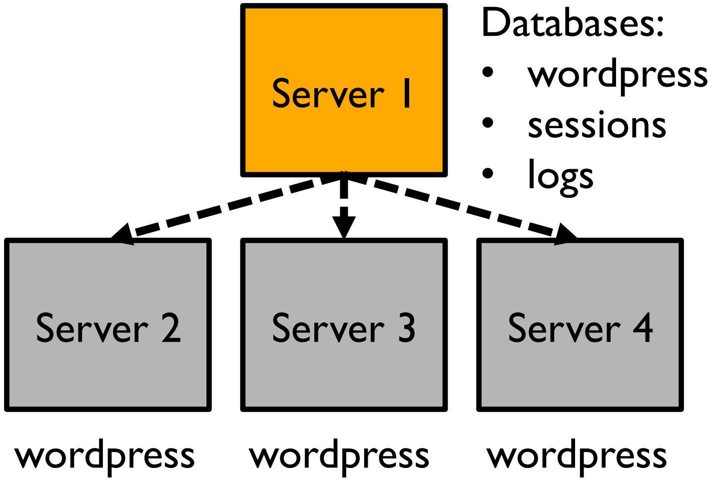
- Tip: Intermediate slaves should use `binlog_format=STATEMENT`.
  - If master sends STATEMENT binlog records, these stay in STATEMENT.
  - If master sends ROW binlog records, these stay in ROW.
  - This configuration is important to support table checksums.

Replication Filters

- Replicate partial data, so slaves handle less traffic.
- http://dev.mysql.com/doc/refman/5.6/en/replication-rules.html

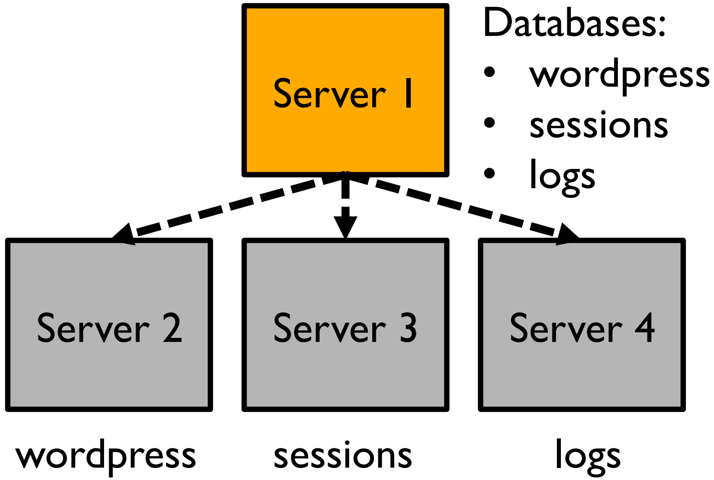
Replication Filter on the Master

- Master writes to its logs for only some databases.
- Then all slaves apply all changes in the logs.



Replication Filter on the Slave

- Master writes changes for all databases to its logs.
- Then each slave downloads all binary logs, but executes only changes against specific databases.



Replication Filter Risks

- Multi-database updates don't work.
- Table checksums must run database-specific.
- Slaves cannot be promoted to master, because they don't have a complete set of databases.

Replication

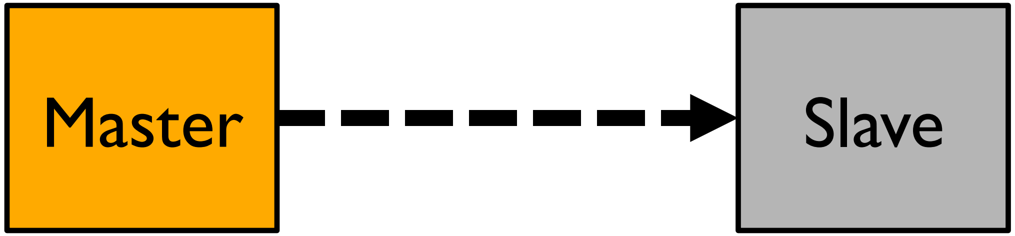
TOPOLOGIES

Replication Topologies

- Master-Slave
- Master-Master
- Tiered-Slave
- Tree
- Master-Master + Tree
- Dual-Tree
- Ring

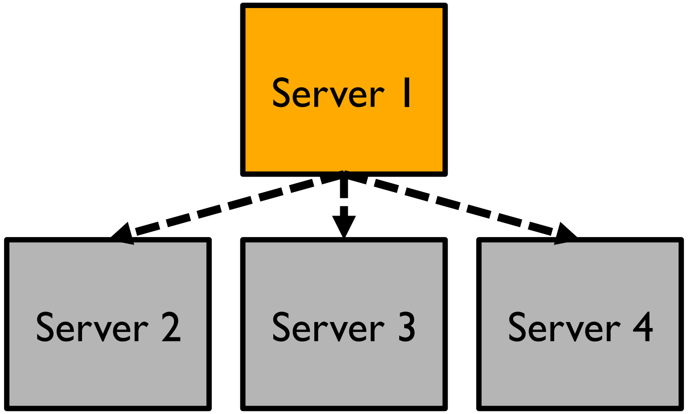
Master-Slave Topology

- Architecture suitable for most projects.
- Use case: slaves for running backups, analytics, or increasing capacity for read queries.
- Doesn't help for failover/failback, or availability during upgrades.



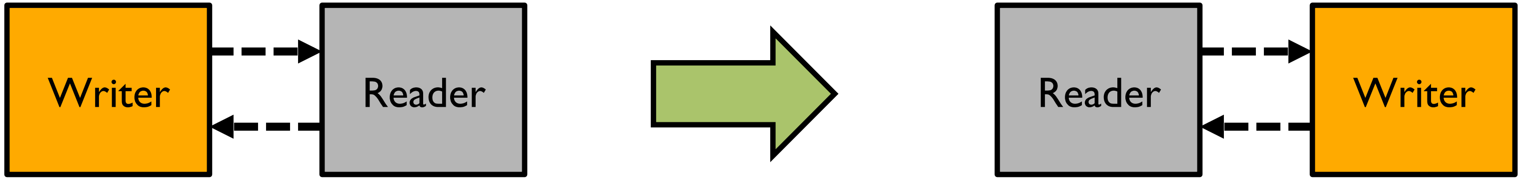
Master-Multiple Slaves Topology

- Use case: additional slaves for other dedicated read-only queries (e.g. reporting), or increasing capacity for read queries.



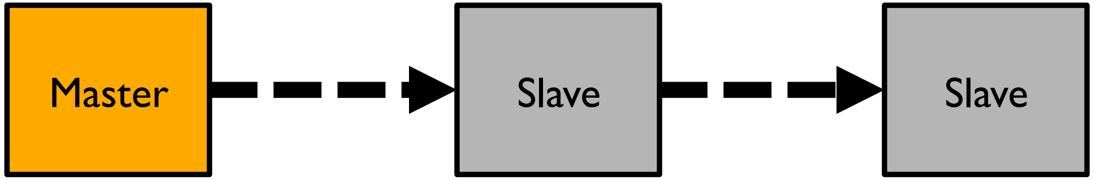
Master-Master Topology

- Use CHANGE MASTER on both MySQL instances to subscribe to changes on the other instance.
- Safest if your applications write to one instance at a time; the other instances are set read-only.
- Use case: failover/failback, availability during upgrades.



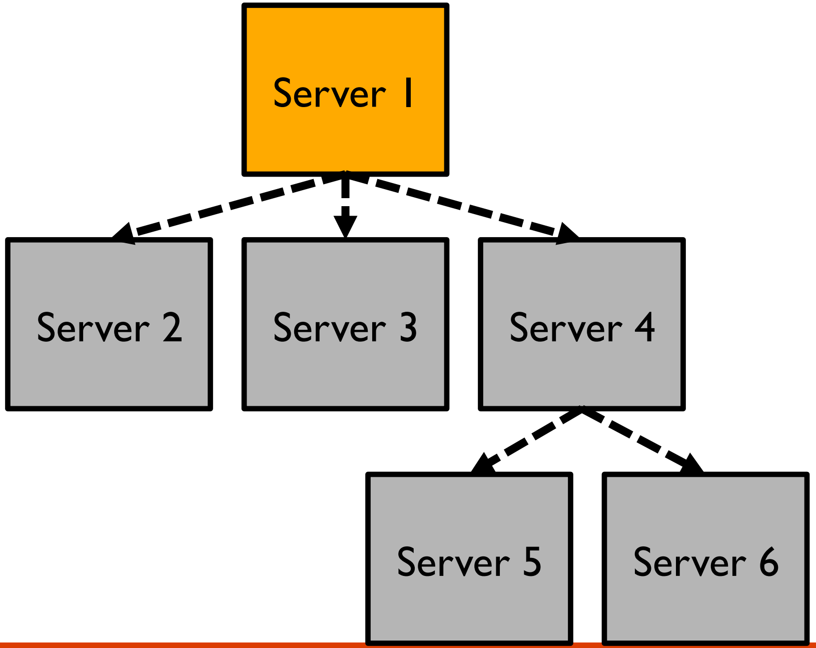
Tiered-Slave Topology

- Avoid multiple slaves downloading binlogs.
- Not a burden for the master, but it costs bandwidth, for example if the slaves are in a remote data center.
- Use case: isolating sets of slaves. E.g.: slaves are in a separate data center. Avoids redundant download of binlogs via the WAN.



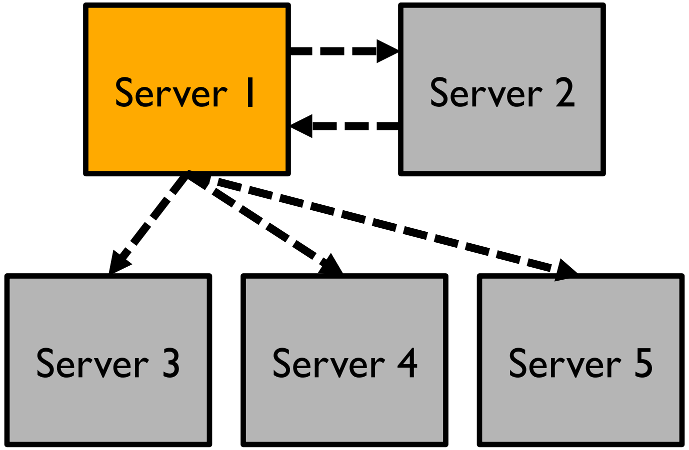
Tree Topology

- Any slave can be a master for “downstream” slaves.
- Use case: mix of read scaling and isolating sets of slaves.



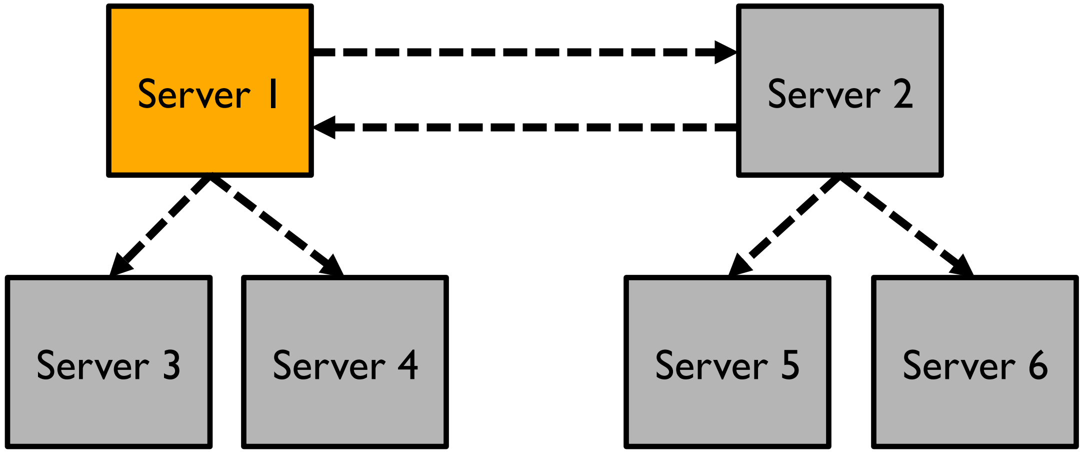
Master-Master + Tree Topology

- One master-master pair, with additional slaves.
- All slaves use a single master to allow the passive master to be freely for maintenance or upgrades.
- Use case: mix of read scaling and failover.



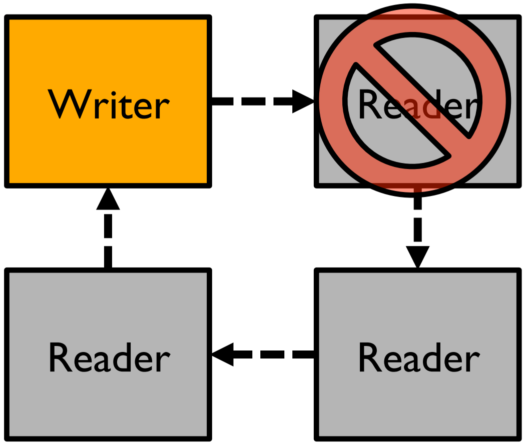
Dual-Tree Topology

- One master-master pair, with additional slaves on each master.
- Use case: mix of read scaling and failover to an alternate data center.



Ring Topology

- Possible, but not recommended.
- Use case: you get increased read capacity, and in theory any slave can take over as master.
- But if any instance fails, all downstream instances stop updating.



Replication

ADMINISTRATION AND MAINTENANCE

Replication Administration and Maintenance

- Starting slave automatically (or not)
- Managing log files
- Monitoring replication health
- Measuring slave lag
- Measuring slave drift
- Correcting slave drift
- Changing masters
- Failover and switchover

Starting Slave Automatically (or not)

- Replication slave threads start automatically, unless you set this in `/etc/my.cnf`:

```
[server]
skip_slave_start = 1
```

- Pros and cons of doing this?
  - Pro: gives the DBA the opportunity to CHANGE MASTER on the slave after startup (change the master, change the binlog coordinates, etc.).
  - Con: requires you to do one more manual step when restarting a slave.

Managing Log Files

- View the current binary logs at any time:

```
mysql> SHOW BINARY LOGS;
+-------------+------------+
| Log_name    | File_size  |
+-------------+------------+
| db1.000023  |        144 |
| db1.000024  |        107 |
+-------------+------------+
```

Managing Log Files

- MySQL creates a new binary log file:
  - When the mysqld server restarts.
  - When the log file size exceeds `max_binlog_size`.
  - When you issue `FLUSH LOGS;`

Managing Log Files

- Manually purge binary logs:

```
mysql> PURGE BINARY LOGS TO 'db1.000024';
```

- Automatically purge binary logs:
  - Percona Server also has an option to purge binary logs when storage exceeds a threshold, instead of by days.

```
[server]
expire_logs_days = 7
```

Managing Log Files

- RESET MASTER
  - Purges all binary logs and initializes master file and position to 1.
- RESET SLAVE
  - Rewrites the slave configuration with default values.
- RESET SLAVE ALL
  - Removes slave configuration completely.

Monitoring Replication Health

- Check for errors:

```
mysql> SHOW SLAVE STATUS\G;
. . .
Slave-IO-Running: Yes
Slave-SQL-Running: No
Last-Errno: 1062
Last-Error: Error 'Duplicate entry '15218'
for key 1' on query. Default database: 'db'.
Query: 'INSERT INTO db.table ( FIELDS )
VALUES ( VALUES )’
. . .
```

Measuring Slave Lag

- One measure of slave lag:

```
mysql> SHOW SLAVE STATUS\G
. . .
Seconds_behind_master: 174
. . .
```

- This is *usually* accurate, but it's really reporting the difference in timestamps between the last executed change by the SQL thread, and the last downloaded change by the IO thread.
- There might be more binary logs on the master that haven't been downloaded yet.

Measuring Slave Lag

- On the master:

```
mysql> REPLACE INTO dummy (timestamp) VALUES
(SYSDATE());
```

- On the slave:

```
mysql> SELECT SYSDATE() - dummy.timestamp FROM
dummy;
```

- This is how Percona Toolkit's pt-heartbeat works.
  - Insert a timestamp into a dummy table once per second. The difference on the slave is always an accurate measure of the real slave lag (within 1 second).
  - http://www.percona.com/doc/percona-toolkit/pt-heartbeat.html

Measuring Slave Drift

- Percona Toolkit's pt-table-checksum
- http://www.percona.com/doc/percona-toolkit/pt-table-checksum.html

Changing Masters

- Making a slave subscribe to a different master:

```
mysql> STOP SLAVE;
mysql> CHANGE MASTER TO
MASTER_HOST='192.168.56.202';
```

- The binary log position of the new master is almost certainly not in sync with old master.
- Discovering the correct binlog coordinate on the new master corresponding to the last change executed on the slave can be tricky.

```
mysql> CHANGE MASTER TO
         MASTER_LOG_FILE='mysql-bin.000123',
         MASTER_LOG_POS=3289439;
mysql> START SLAVE;
```

Failover and Switchover

- Failover is when the current master fails and one of the slave is assigned to become the new master in an automatic, unattended manner.
  - This is even harder than it sounds to automate!
- Switchover is also assigning another server as a new master, but in a planned manner.
  - This is much more achievable, if you can stop application traffic even for a few seconds.

Exercises

- Create an error by CREATE TABLE only on the slave, then create the same table on the master. What is the error?
- Issue a long-running update on the master and let it propagate to the slave. How is the lag reported?
- Run pt-table-checksum. Change some data on the slave, and run pt-table-checksum again.

REPLICATION

PROBLEMS AND SOLUTIONS

Replication Problems and Solutions

- Slave lag
- Slave drift
- Data corruption
- Non-deterministic changes
- Out of band changes
- Bad server ids
- Non-replicated data
- Risks of dual-masters
- Logs out of sync
- Oversized packets
- Limited bandwidth
- Disk space exhaustion
- Lost events

Slave Lag

- Occasional slave lag is a fact of life, but sometimes it can get out of control.
- Mitigation of slave lag:
  - Faster CPU to execute SQL statements more quickly.
  - Faster I/O system to write changes more quickly.
  - Use `binlog_format=ROW` if the SQL statements are slow to execute.
  - Replicate fewer changes to slaves (replication filtering).
  - Balance writes over multiple master-slave pairs (sharding).
  - Pre-warm buffer pool on the slave so updates run faster.

Slave Drift

- Percona Toolkit's pt-table-sync
- http://www.percona.com/doc/percona-toolkit/pt-table-sync.html

Data Corruption

- If the slave drift is too severe, it's often a quicker and simpler operation to *reinitialize the slave*:
  - STOP SLAVE;
  - Drop all the databases (once we've decided they're too far gone to be useful anyway).
  - Acquire a fresh backup from the master, or from another slave.
  - Restore the backup to reinitialize the damaged slave.
  - CHANGE MASTER to the right binlog coordinate.
  - START SLAVE;

Non-Deterministic Changes

- SQL statements may change data differently on the slave than on the master.
- Examples:

```
UPDATE … ORDER BY RAND() LIMIT 1;
INSERT INTO table (pk) VALUES (UUID());
UPDATE … WHERE ts > SYSDATE();
```

Out of Band Changes

- Some misbehaving applications (or misbehaving users) may change data directly on the slave.
- Mitigation strategy:
  - Enable the `read_only` option for all instances except the primary master.

```
mysql> SET GLOBAL read_only=1;
```

  - The root user and the replication SQL thread can still make changes.

Bad Server Ids

- Misconfiguration of `server_id` can prevent replication from running:

```
mysql> START SLAVE;
ERROR 1200 (HY000): The server is not
configured as slave; fix in config file or
with CHANGE MASTER TO
```

- Mitigation strategy: As the error suggests, set `server_id` and restart the instance.

Non-Replicated Data

- Some changes depend on data that doesn't exist on the slave.
  - Temporary tables.
  - Replication-filtered tables.
- Mitigation strategies:
  - Avoid using temp tables as a source for hybrid read/write operations (e.g. `INSERT…SELECT`, multi-table `UPDATE/DELETE`, etc.).
  - Use ROW-based replication.

Risks of Dual Masters

- Since replication is asynchronous, your applications may change data on two masters simultaneously, introducing a consistency violation that isn't caught until the changes propagate.
  - E.g., duplicate key violations.

Risks of Dual Masters

- Mitigation strategies:
  - Write to one master at a time. Make the other `read_only`.
  - Let applications write changes to both masters, but be careful to write only to one instance or the other for a given subset.
  - Configure each instance so that one allocates odd values, and the other one allocates even values. E.g. in `/etc/my.cnf`:

```
[server]
auto_increment_increment=2
auto_increment_offset=N
```

Logs Out of Sync

- Errors in downloading binary logs can stop replication and report an error:

```
Last_IO_Error: Got fatal error 1236 from
master when reading data from binary log:
'Could not find first log file name in binary
log index file'
```

- For example, the slave was stopped for a long time, and when it resumed, the binary log file it had been reading had been purged on the master.

Oversized Packets

- Large data payloads (e.g. large BLOB/TEXT data) can be too large for the default packet size limit.
- The default is 4MB*; the maximum is 1GB.
- Mitigation strategy: increase this configuration setting in `/etc/my.cnf` on *both* master and slave:

```
[server]
max_allowed_packet = 100M
```

- MySQL 5.1.64, 5.5.26, 5.6.6 introduces new variable `slave_max_allowed_packet`, default value 1GB.

* default max_allowed_packet is 1MB in MySQL < 5.6

Disk Space Exhaustion

- The master can run out of disk space as binary logs accumulate, even if the database isn't large.
- The slave can run out of disk space by downloading binary logs.
- Mitigation strategies:
  - Provision disks liberally, with plenty of space to spare. Don't run at 90%+ disk full.
  - Set up tools to alert you when disk space is running out.
  - Use `expire_logs_days` and `PURGE BINARY LOGS` to free disk space as needed.

Lost Events

- Any GRANT statement that fails on the master causes replication to stop:

```
Last_Errno: 1590
Last_Error: The incident LOST_EVENTS occured
on the master. Message: error writing to the
binary log
```

- Skip the GRANT statement on all slaves, and restart replication.

http://bugs.mysql.com/bug.php?id=68892

Replication

ENSURING DATA INTEGRITY WITH PERCONA TOOLKIT

Data Drift

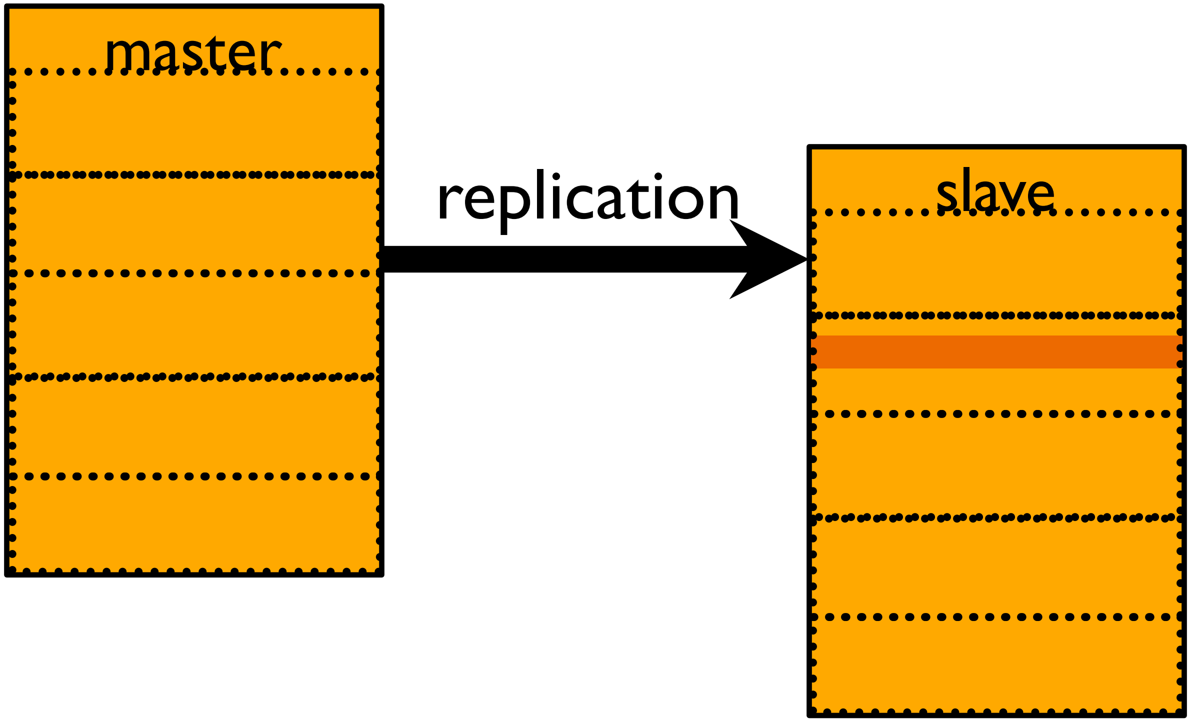
- MySQL slaves may not be perfect replicas.
  - Non-deterministic statements.
  - Out-of-band changes directly on the slave.
  - Slave may lag and fail to keep up.
  - No built-in checking.
  - Are you using a slave for backups or reporting?

pt-table-checksum

- Perform an online replication consistency check, or checksum MySQL tables efficiently.
- This is the solution to *detect* data drift.
- Calculates checksums against "chunks" of rows.
- The calculation propagates to slaves.

http://www.percona.com/doc/percona-toolkit/pt-table-checksum.html

Visualize This



Example

```
$ pt-table-checksum
            TS ERRORS  DIFFS     ROWS  CHUNKS SKIPPED    TIME TABLE
12-01T11:00:13      0      0   633135       7       0   3.814 imdb.aka_name
12-01T11:00:15      0      0   290859       1       0   1.682 imdb.aka_title
Checksumming imdb.cast_info:  24% 01:34 remain
Checksumming imdb.cast_info:  48% 01:03 remain
Checksumming imdb.cast_info:  75% 00:28 remain
12-01T11:02:13      0      0 22187768     163       0 118.059 imdb.cast_info
12-01T11:02:25      0      0  2406561      20       0  12.292 imdb.char_name
12-01T11:02:25      0      0        4       1       0   0.123 imdb.comp_cast_type
12-01T11:02:27      0      0   241457       1       0   1.291 imdb.company_name
12-01T11:02:27      0      0        4       1       0   0.033 imdb.company_type
12-01T11:02:27      0      0    97304       1       0   0.492 imdb.complete_cast
12-01T11:02:27      0      0      113       1       0   0.079 imdb.info_type
12-01T11:02:28      0      0    87520       1       0   0.367 imdb.keyword
12-01T11:02:28      0      0        7       1       0   0.027 imdb.kind_type
12-01T11:02:28      0      0       18       1       0   0.030 imdb.link_type
12-01T11:02:37      0      0  1965016      15       0   9.142 imdb.movie_companies
Checksumming imdb.movie_info:  64% 00:16 remain
12-01T11:03:34      0      0  9748370      76       0  57.105 imdb.movie_info
12-01T11:03:38      0      0   934655       8       0   4.026 imdb.movie_info_idx
12-01T11:03:49      0      0  2776445      15       0  10.552 imdb.movie_keyword
12-01T11:03:52      0      0   922518       7       0   3.051 imdb.movie_link
12-01T11:04:07      0      0  2812743      25       0  15.817 imdb.name
12-01T11:04:29      0      0  2271731      22       0  21.495 imdb.person_info
12-01T11:04:29      0      0       12       1       0   0.015 imdb.role_type
12-01T11:04:39      0      0  1543719      17       0  10.189 imdb.title
```

Let's Break It

- Delete 5% of data on the slave:

```
mysql> DELETE FROM title
WHERE RAND()*100 < 5;
Query OK, 77712 rows affected (2.09 sec)
```

Re-check

```
$ pt-table-checksum --tables imdb.title

            TS ERRORS  DIFFS     ROWS  CHUNKS SKIPPED    TIME TABLE
12-03T05:04:26      0     14  1543719      16       0  10.512 imdb.title
```

Check the Slave(s)

```
mysql> SELECT db, tbl, SUM(this_cnt) AS total_rows, COUNT(*) AS chunks
   FROM percona.checksums
   WHERE master_cnt <> this_cnt
     OR master_crc <> this_crc
     OR ISNULL(master_crc) <> ISNULL(this_crc)
   GROUP BY db, tbl;
+------+-------+------------+--------+
| db   | tbl   | total_rows | chunks |
+------+-------+------------+--------+
| imdb | title |    1466007 |     14 |
+------+-------+------------+--------+
```

pt-table-sync

- Synchronize MySQL table data efficiently.
- This is the solution to *correct* data drift.

http://www.percona.com/doc/percona-toolkit/pt-table-sync.html

Method 1: Sync Master to Slave(s)

```
$ pt-table-sync --verbose --execute --replicate percona.checksums huey

# Syncing via replication h=192.168.56.112
# DELETE REPLACE INSERT UPDATE ALGORITHM START    END      EXIT DATABASE.TABLE
#      0      47      0      0 Chunk     05:05:46 05:05:47 2    imdb.title
#      0     795      0      0 Chunk     05:05:47 05:05:49 2    imdb.title
#      0    5070      0      0 Chunk     05:05:49 05:06:01 2    imdb.title
#      0    6361      0      0 Chunk     05:06:01 05:06:16 2    imdb.title
#      0    6867      0      0 Chunk     05:06:16 05:06:36 2    imdb.title
#      0    7297      0      0 Chunk     05:06:36 05:06:55 2    imdb.title
#      0    7504      0      0 Chunk     05:06:55 05:07:13 2    imdb.title
#      0    7688      0      0 Chunk     05:07:13 05:07:34 2    imdb.title
#      0    7346      0      0 Chunk     05:07:34 05:07:52 2    imdb.title
#      0    7065      0      0 Chunk     05:07:52 05:08:10 2    imdb.title
#      0    6937      0      0 Chunk     05:08:10 05:08:27 2    imdb.title
#      0    6695      0      0 Chunk     05:08:27 05:08:43 2    imdb.title
#      0    6765      0      0 Chunk     05:08:43 05:09:00 2    imdb.title
#      0    1275      0      0 Chunk     05:09:00 05:09:04 2    imdb.title
```

Method 2: Sync Slave to Master

```
$ pt-table-sync --verbose --execute --sync-to-master h=dewey,D=imdb,t=title

# Syncing D=imdb,P=5528,h=127.0.0.1,p=...,t=title,u=root
# DELETE REPLACE INSERT UPDATE ALGORITHM START    END      EXIT DATABASE.TABLE
#      0   23097      0      0 Chunk     16:07:21 16:08:21 2    imdb.title
```

Method 3: Sync Two Hosts

- pt-table-sync won't let you clobber a slave by syncing it to some host other than its master.

```
$ pt-table-sync --verbose
  --execute h=huey d=dewey
  --tables imdb.title
```

```
Can't make changes on h=dewey because it's a slave. See the documentation
section 'REPLICATION SAFETY' for solutions to this problem. at /usr/bin/pt-
table-sync line 10642.
```

Method 3: Sync Two Hosts

- Now let's try again, after running `RESET SLAVE`.

```
$ pt-table-sync --verbose
--execute h=huey h=dewey
--tables imdb.title
```

```
# Syncing h=dewey
# DELETE REPLACE INSERT UPDATE ALGORITHM START    END      EXIT DATABASE.TABLE
#      0       0  30867      0 Chunk     13:33:27 13:35:28 2    imdb.title
```

Security

MYSQL PRIVILEGE SYSTEM

MySQL Passwords

- In MySQL authentication protocol, passwords are not sent as plaintext.
  – This is not true for MySQL's pluggable authentication like PAM—make sure to use SSL.

MySQL Passwords

- MySQL 5.6 passwords may be stored as a SHA-256 hash with salt.
- MySQL 4.1-5.5 passwords are stored as a double-SHA1 hash.
- MySQL 4.0 and earlier passwords were DES-encrypted.
  - Some sites still have `old_passwords=1` even if they now use a more recent version of MySQL.

MySQL Passwords

- Password expiration
- Password strength
- Disabling accounts

Granting Privileges

```
1.  mysql> GRANT SELECT, UPDATE on database.table TO
2.      -> 'user'@'hostname' IDENTIFIED BY 'password';
```

- There is no concept of object ownership. You just grant a series of permissions based on a pattern.
- No support for SQL roles or user groups.
- Username and host *combined* grants access. It's possible to have different permissions based on where you access from.

http://dev.mysql.com/doc/mysql-security-excerpt/5.6/en/privileges-provided.html

More Examples

```
1.  mysql> GRANT replication slave ON *.* TO
2.         -> 'repl'@'192.168.1.1' IDENTIFIED BY 'swordfish';
3.
4.  mysql> GRANT SELECT, CREATE TEMPORARY TABLES ON imdb.*
5.         -> TO 'webapp'@'192.168.1.%' IDENTIFIED BY 'swordfish';
```

- Column level privileges also exist - but are not recommended. Full list of privileges:
  - http://dev.mysql.com/doc/refman/5.6/en/grant.html

Hardware and Operating Systems

Percona Training

http://www.percona.com/training

Common Questions

- Virtualization (cloud) or bare metal servers?
- SSD or not SSD?
- One big machine, or a few small machines?
- Debian or Red Hat?
- Raid 5 or RAID 10?
- Filesystems?
- Kernel / OS Settings?

Virtualization

- Not entirely a technical question.
- Many cloud environments have limited hardware choices, with the higher power options very expensive:
  - On Amazon EC2, the most memory you can have is 68.4G for $1752/month or $1144 reserved for 12 months.

Virtualization (cont.)

- There are a range of technical problems best solved via hardware.
  - If one [large] machine could do the work, it normally* does not make sense to make changes to software to work with 10 smaller nodes.
  - For many customers "cloud = agility." This is not true when unnecessary complexity reduces agility.

* Clarified in a few slides time!

Bare Metal Servers

- Newer Nehalem servers have up to 64 memory slots.
  - That's 1TB memory using 16GB DIMMs or 256GB using the cheaper 4GB DIMMs.
- For IO, there are flash PCI cards which are capable of 10K + IOPS.
  - A hard drive might be capable of 100-200 IOPS.

There is no technical restriction from using flash or 256GB memory in a virtualized machine. It's just not common in cloud hosting.

Bare Metal (cont.)

- Simple can be better.
- You reliably know your minimum performance.
  - You can reliably tell that a gigabit ethernet link is yours alone.
  - You can size settings like `innodb_io_capacity` to use "all free capacity." I.e., you don't care if this results in more IO, you would rather all available capacity be used.
  - In practice this can make debugging problems much easier, due to less *unknowns*.

Many of our customers can not tell if it was last week's deployment that suddenly made the application slow, or other users on the same system being more active.

One Big Machine or Many Small?

- Depends on the goal:
  - One large machine is the easiest to deploy and manage.
  - Many small servers can be helpful for the purposes of isolation (based on either task or customer).
- Examples of isolation:
  - Create slaves for reporting queries.
  - Create slaves which are used by Sphinx for fulltext indexing.

If your customers pay $1000 month each for a SAAS application, they often have a certain "expectation" of performance.

SSD or Not?

- There are some very strong SSD options already available. Many customers are using them in production.
- Understanding if it is the best choice is workload dependent. i.e.
  - Reads vs Writes / Memory Fit?
  - Need for better response or throughput?

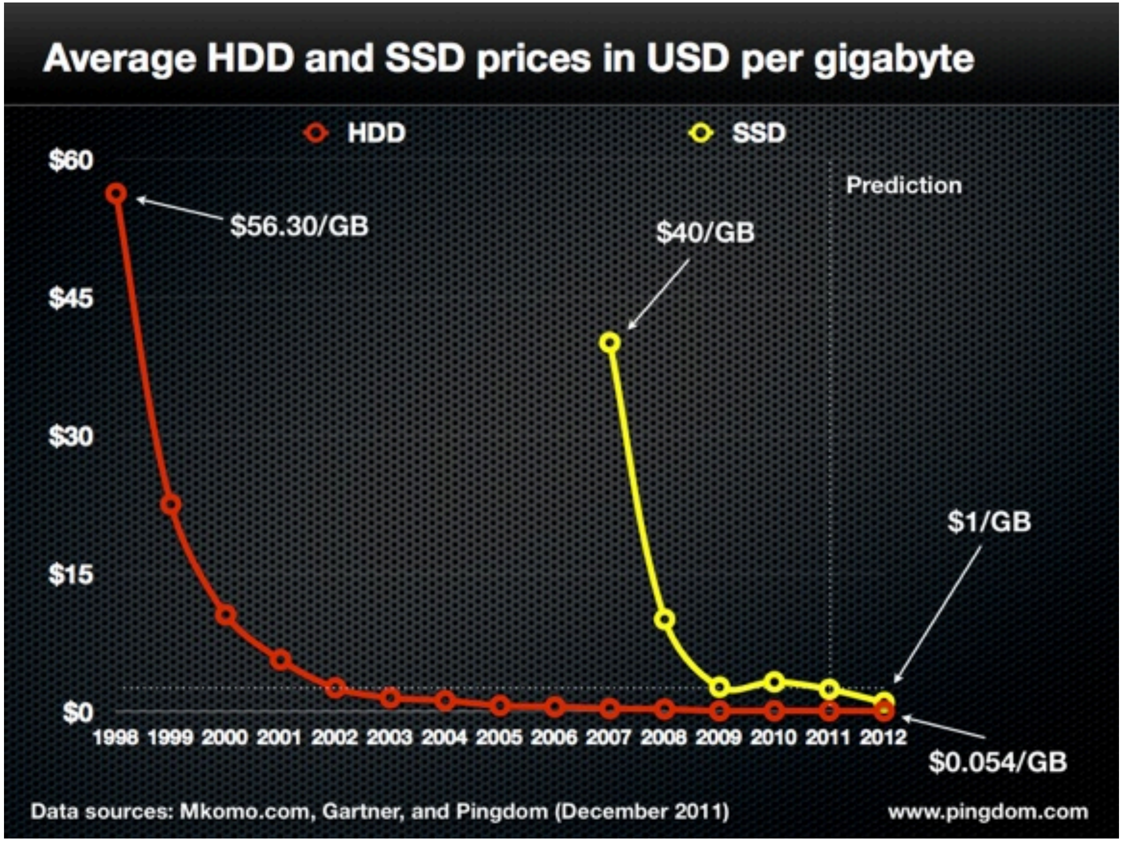
Advantage of SSDs (1)

- Fast access time means that the cache miss path is far less expensive. i.e.
  - ~10ms changes to less than 1ms per IO.
- This means that:
  - Some applications may not need nearly as high cache hit ratios. A 50ms page load can barely afford any cache misses.
  - Some operational issues become easier, such as a reduced warm up time post restart.

Advantage of SSDs (2)

- Throughput of SSDs is much cheaper than hard drives, even if *storage* costs more.
- Many can do 10K+ IOPS.

Price/Size Ratio



Debian or Red Hat (etc)?

- Tends not to matter much.
- What matters the most is that the release is supported for the duration of time the server will be deployed.
  - Fedora, Gentoo, Ubuntu (non LTS) are likely not good choices for this reason.

RAID5 or RAID10? (1)

- Workload specific—most likely answer is RAID10.
  - If cache fit is large enough, reads can be nearly eliminated, and writes are more an issue.
  - With RAID5 if you do not write a full stripe, you need to *read before you write* to recalculate parity.
  - For sequential writes only, RAID5 may perform better for the same number of disks.

RAID5 or RAID10? (2)

- Always difficult to answer this question with 100% confidence.
  - RAID controller vendors provide little transparency into internal operations.
  - Just because an optimization could theoretically apply, does not guarantee that it "does apply."

Stripe Size?

- Similar difficulty to answer reliably.
- What probably matters most is:
  - What is the vendor default?
  - Are you ever writing across stripe boundaries?

Stripe Size (cont.)

- What is the vendor default?
  - Likely has the most optimizations. Any changes need to be verified.
- Are you ever writing across stripe boundaries?
  - InnoDB almost always writes 16K at a time.
  - If you have a 16K stripe, but InnoDB pages are non-aligned each write will be on two stripes.
  - Aligning can be difficult[1].
  - Some customers choose larger stripe sizes to"amortize" these boundary-writes.

http://thunk.org/tytso/blog/2009/02/20/aligning-filesystems-to-an-ssds-erase-block-size/

Other RAID Controller Tips

- You want to purchase the battery option.
- This allows you to configure caches to write-back mode:
  - Performance from the application on fsync is very good.
  - Data durability is still available.
  - Merging can happen on the RAID controller before writing down to the physical disks.

Filesystems

- Use XFS when using multiple disks.
  - Supports better concurrency.
- ext3 will serialize a write to an individual file.
  - It can not take advantage of InnoDB multiple write threads effectively.
  - A fsync operation is also serialized.

Kernel and OS Settings

- Mount filesystems with `noatime`
- Set `vm.swappiness = 0` in /etc/sysctl.conf
- *[With RAID]* Change the IO schedulers from the default to either `deadline` or `noop`.
  - Check with: `cat /sys/block/DEVICE/queue/scheduler`
  - Change this persistently in `/etc/grub.conf`

Possible Networking Wins

- It might be worth increasing `/proc/sys/net/ipv4/ip_local_port_range` to get more local TCP/IP ports available if handling a lot of connections.
- Decreasing the value of `/proc/sys/net/ipv4/tcp_fin_timeout` can help you reduce the time it takes to idle-recycle a connection.
  - Technically it breaks the standard, but should work fine on a local network.

Query Optimization

Percona Training

http://www.percona.com/training

Table of Contents

Query Optimization

QUERY PLANNING

About This Chapter

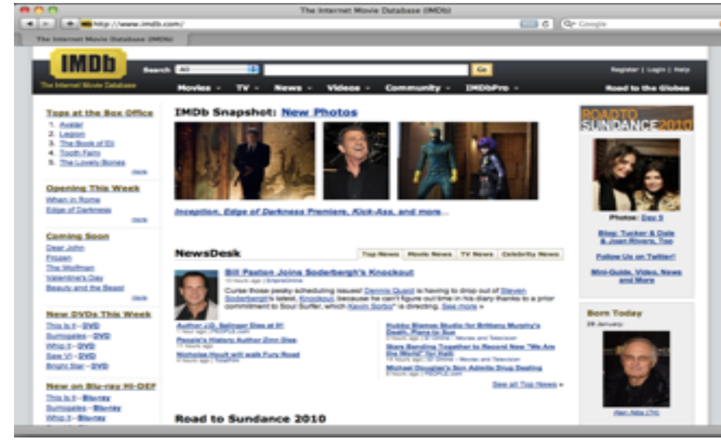
- The number one goal is to **have faster queries**.
- The process is:
  - We first ask MySQL what its intended execution plan is.
  - If we don't like it, we make a change, and try again...

It All Starts with EXPLAIN

- Bookmark this manual page:
  http://dev.mysql.com/doc/refman/5.6/en/explain-output.html
- It is *the best source* for anyone getting started.

Example Data

- IMDB database loaded into InnoDB tables (~5GB).
- Download it and import it for yourself using imdbpy2sql.py: http://imdbpy.sourceforge.net/



First Example

```
CREATE TABLE title (
  id              int            NOT NULL AUTO_INCREMENT,
  title           text           NOT NULL,
  imdb_index      varchar(12)    DEFAULT NULL,
  kind_id         int            NOT NULL,
  production_year int            DEFAULT NULL,
  imdb_id         int            DEFAULT NULL,
  phonetic_code   varchar(5)     DEFAULT NULL,
  episode_of_id   int            DEFAULT NULL,
  season_nr       int            DEFAULT NULL,
  episode_nr      int            DEFAULT NULL,
  series_years    varchar(49)    DEFAULT NULL,
  title_crc32     int(10)        unsigned DEFAULT NULL,
  PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Find the Title Bambi

3.09s

```
mysql> EXPLAIN SELECT id,title,production_year FROM title
       WHERE title = 'Bambi' ORDER BY production_year\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1535171
        Extra: Using where; Using filesort
1 row in set (0.00 sec)
```

ALL means tablescan

In this case a sort is required because of the ORDER BY

Anticipated number of rows to be examined

Additional filtering may be possible before passing to sort.

Aha! Now Add an Index

```
1.  mysql> ALTER TABLE title ADD INDEX (title);
2.  ERROR 1170 (42000): BLOB/TEXT column 'title'
3.  used in key specification without a key length

4.  mysql> ALTER TABLE title ADD INDEX (title(50));
5.  Query OK, 0 rows affected (8.09 sec)
6.  Records: 0 Duplicates: 0 Warnings: 0
```

We Must Revisit

0.00s

```
mysql> EXPLAIN SELECT id,title,production_year FROM title
        WHERE title = 'Bambi' ORDER by production_year\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ref
possible_keys: title
          key: title
      key_len: 152
          ref: const
         rows: 4
        Extra: Using where; Using filesort
1 row in set (0.00 sec)
```

Using = for comparison, but not primary key lookup.

Identified title as a candidate index, chose to use it.

Size of the index used (in bytes)

Anticipated number of rows to be examined dropped considerably.

0.00s

Other Ways of Accessing

```
mysql> EXPLAIN SELECT id,title,production_year FROM title
        WHERE id = 55327\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
        Extra: NULL
1 row in set (0.00 sec)
```

At most one matching row.

In InnoDB the primary key is often much faster than all other keys.

0.00s

LIKE

```
mysql> EXPLAIN SELECT id,title,production_year FROM title
        WHERE title LIKE 'Bamb%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: range
possible_keys: title
          key: title
      key_len: 152
          ref: NULL
         rows: 98
        Extra: Using where
1 row in set (0.00 sec)
```

Type is range. BETWEEN, IN() and < > are also ranges.

Number of rows to be examined has increased - we are not specific enough.

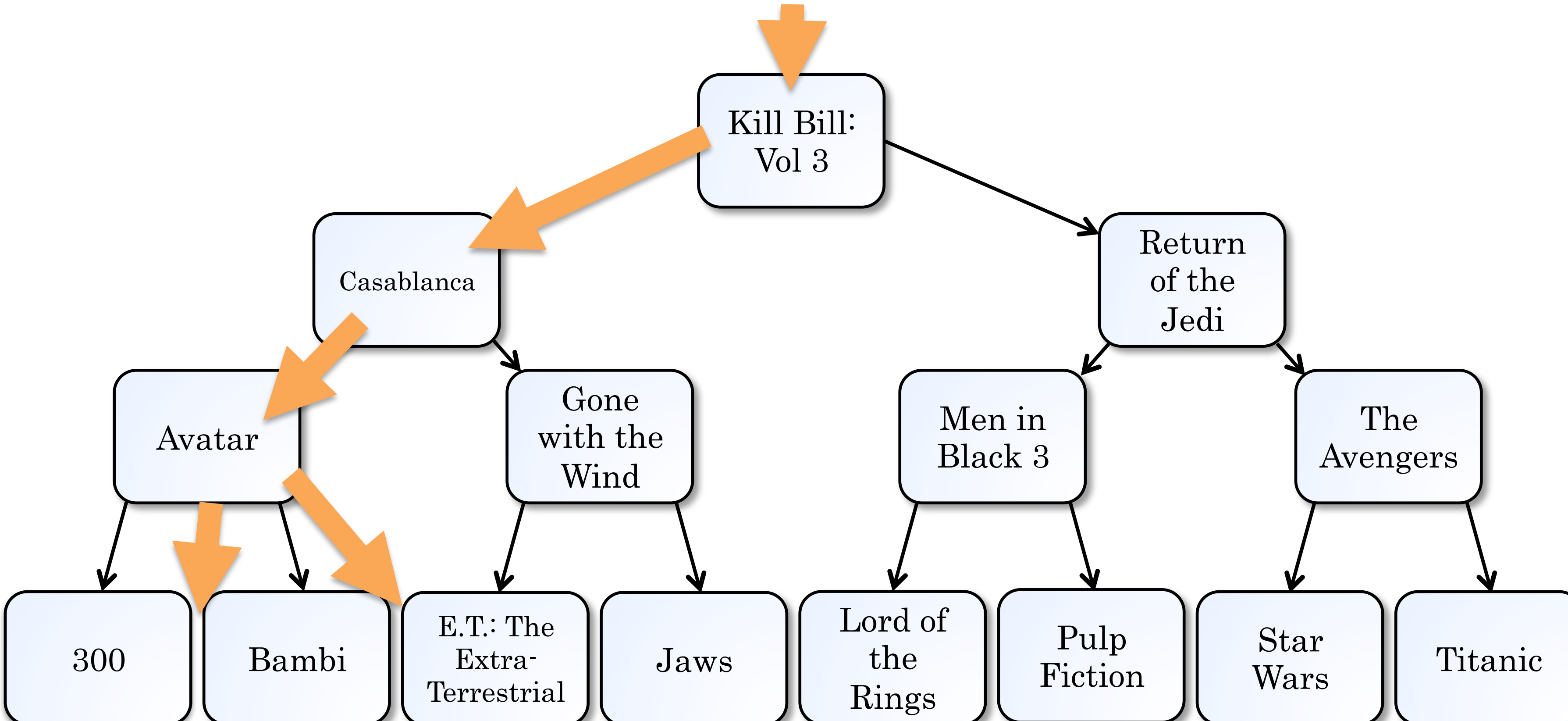
Ignore the time with EXPLAIN. Only look at the time for a query.

Why's That a Range?

- We're looking for titles between Bamb*A* and Bamb*Z**
- When we say index in MySQL, we mean trees.
  - That is, B-Tree/B+Tree/T-Tree.
  - Pretend they're all the same (for simplification).
  - There is only radically different indexing methods for specialized uses: MEMORY Hash, FULLTEXT, spatial or 3 party engines.

* In reality the range is a little wider

What's That?

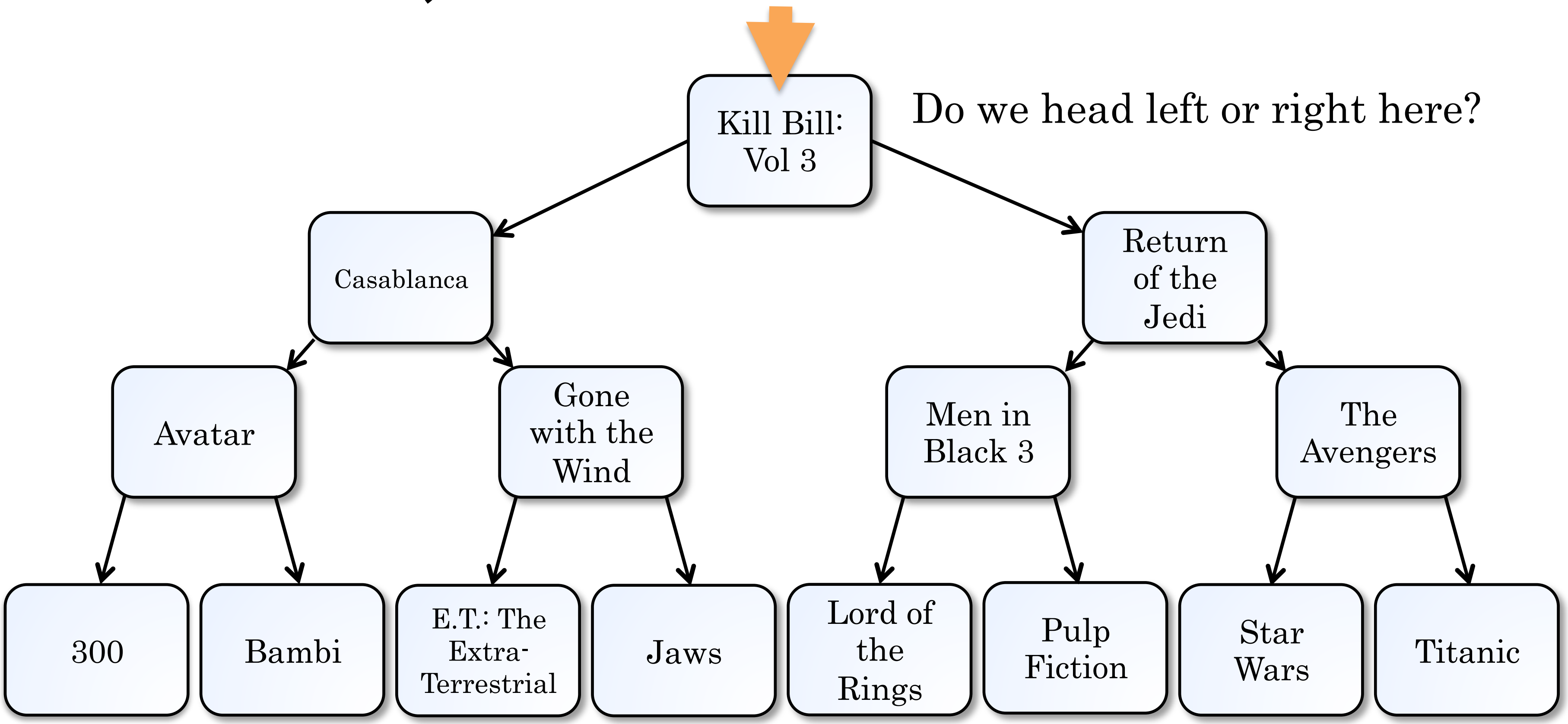

Kill Bill: Vol 3
Casablanca
Return of the Jedi
Avatar
Gone with the Wind
Men in Black 3
The Avengers
300
Bambi
E.T.: The Extra-Terrestrial
Jaws
Lord of the Rings
Pulp Fiction
Star Wars
Titanic

Could This Be a Range?

3.2s

```
mysql> EXPLAIN SELECT id,title,production_year FROM title
       WHERE title LIKE '%ulp Fiction'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1442263
        Extra: Using where
1 row in set (0.00 sec)
```

No, We Can't Traverse



LIKE 'Z%'

0.05s

```
mysql> EXPLAIN SELECT id,title,production_year FROM title
       WHERE title LIKE 'Z%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: range
possible_keys: title
          key: title
      key_len: 77
          ref: NULL
         rows: 13718
        Extra: Using where
1 row in set (0.00 sec)
```


LIKE 'T%'

3.13s

```
mysql> EXPLAIN SELECT id,title,production_year FROM title
       WHERE title LIKE 'T%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ALL
possible_keys: title
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1442263
        Extra: Using where
1 row in set (0.00 sec)
```

LIKE 'The %'

3.07s

```
mysql> EXPLAIN SELECT id,title,production_year FROM title
       WHERE title LIKE 'The %'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ALL
possible_keys: title
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1442263
        Extra: Using where
1 row in set (0.00 sec)
```

MySQL Is *Reasonably* Smart

- It dynamically samples the data to choose which is the better choice—or in some cases uses static statistics.*
- This helps the optimizer choose:
  - Which indexes will be useful.
  - Which indexes should be avoided.
  - Which is the better index when there is more than one.

* To refresh statistics run `ANALYZE TABLE` *`table_name`*;

Why Avoid Indexes?

- B-Trees work like humans search a phone book;
  - Use an index if you want just a few rows.
  - Scan cover-to-cover if you want a large percentage.

Why Avoid Indexes (cont.)

- Benchmark on a different schema (lower is better):

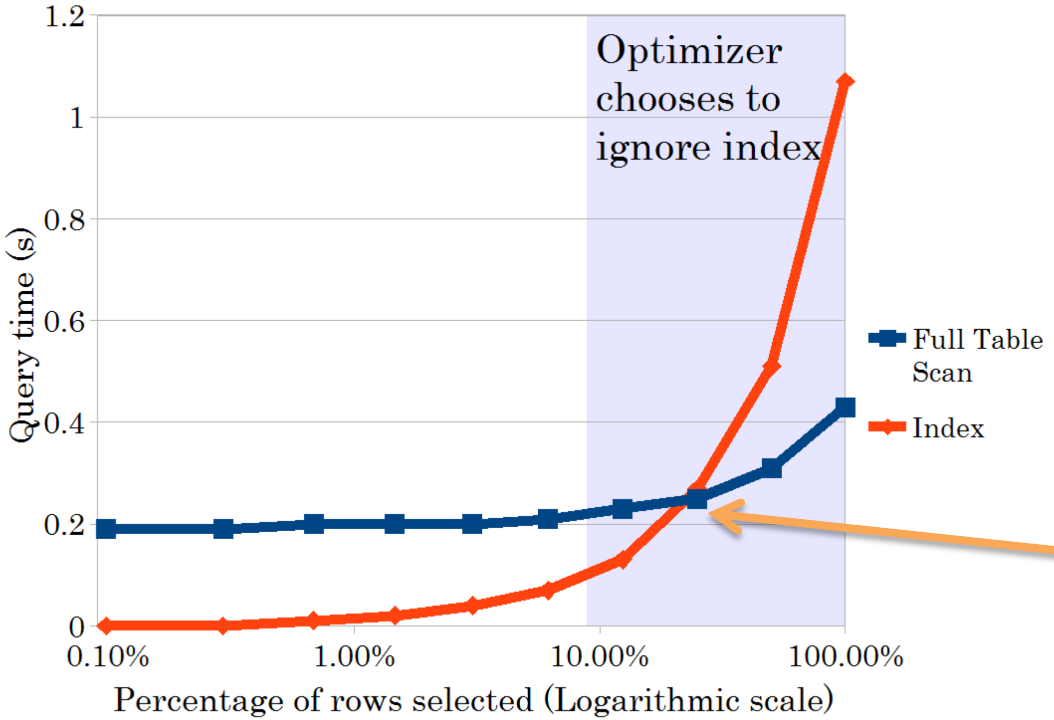


Table scan has a relatively fixed cost (blue line).

The index has completely different effectiveness depending on how much it can filter.

Hopefully MySQL switches at the right point (it does it a bit early in this case).

What You Should Take Away

- **Data is absolutely critical.**
  - Development environments should contain sample data exported from production systems.
  - A few thousands of rows is usually enough for the optimizer to behave like it does in production.

What You Should Take Away (cont.)

- **Input values are absolutely critical.**
  - Between two seemingly identical queries, execution plans may be *very* different.
  - Just like you test application code functions with several values for input arguments.

http://www.mysqlperformanceblog.com/2009/10/16/how-not-to-find-unused-indexes/

Query Optimization

EXPLAINING THE EXPLAIN

How to Explain The EXPLAIN

- The tables are read in the **order** displayed by `EXPLAIN`
- The **id** column is a sequential identifier of `SELECT` statements in the query
- The **select_type** column indicates type of `SELECT` (simple, primary, subquery, union, derived, ...)
- The **type** column says which join type will be used
- The **possible_keys** column indicates which indexes MysQL can choose from use to find the rows in this table
- The **key** column indicates which index is used

How to Explain The EXPLAIN (cont)

- **key_len** tells the length of the key that was used (important to find which parts of a composite index are used)
- **ref** shows which columns or constants are compared to the index named in **key** column to select rows from the table
- **rows** says how many rows have to be examined in order to execute each step of the query (the product of all **rows** columns is the total number of rows that must be examined to solve the query)
- **Extra** contains additional information about how MySQL resolves the query (see http://dev.mysql.com/doc/refman/5.6/en/explain-output.html#explain-extra-information)

Types in EXPLAIN

- The following slides show possible values for `EXPLAIN type`, ordered (approximately) from the fastest to the slowest
  - `FULLTEXT` access type (and its special indexes) are not covered on this section

NULL

- Not really a plan: no data is returned
- See 'Extra' for a reason

```
mysql> EXPLAIN SELECT * FROM title WHERE 1 = 2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: NULL
         type: NULL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
        Extra: Impossible WHERE
1 row in set (0.00 sec)

mysql> EXPLAIN SELECT * from title where id = -1\G
         type: NULL
        Extra: Impossible WHERE noticed after reading const tables
```

Made internally equivalent to SELECT NULL WHERE 0;

system

- The table has only one row (=system table)
- A seldom used special case of the const joint type

```
mysql> EXPLAIN SELECT id FROM (SELECT * FROM title LIMIT 1) AS one\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: <derived2>
         type: system
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1
        Extra: NULL
*************************** 2. row ***************************
           id: 2
  select_type: DERIVED
        table: title
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1396980
        Extra: NULL
2 rows in set (0.00 sec)
```

const

- Used when comparing a literal with a non-prefix PRIMARY/UNIQUE index
- The table has at the most one matching row, which will be read at the start of the query
- Because there is only one row, the values can be regarded as constants by the optimizer
- This is very fast since table is read only once

const (cont.)

```
mysql> EXPLAIN SELECT * FROM title WHERE id = 55327\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
        Extra: NULL
1 row in set (0.00 sec)
```

eq_ref

- **One row** will be read from this table for each combination of rows from the previous tables
- The **best possible** join type (after *const*)
- Used when the whole index is used for the = operator with a `UNIQUE` or `PRIMARY KEY`

eq_ref (cont.)

```
mysql> EXPLAIN SELECT title.title, kind_type.kind FROM kind_type JOIN title
       ON kind_type.id = title.kind_id WHERE title.title = 'Bambi'\G
```

```
*************** 1. row ***************
           id: 1
  select_type: SIMPLE
        table: title
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1396980
        Extra: Using where
```

```
*************** 2. row ***************
           id: 1
  select_type: SIMPLE
        table: kind_type
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: imdb.title.kind_id
         rows: 1
        Extra: NULL
2 rows in set (0.00 sec)
```

Can you think of a way of improving this query?

ref

- **Several rows** will be read from this table for each combination of rows from the previous tables
- Used if the join uses only a left-most prefix of the index, or if the index is not `UNIQUE` or `PRIMARY KEY`
- Still **not bad**, if the index matches only few rows

ref (cont.)

```
mysql> EXPLAIN SELECT distinct t1.title FROM title t1 JOIN title t2
       WHERE t1.title = t2.title and t1.id <> t2.id\G
*************** 1. row ***************
           id: 1
  select_type: SIMPLE
        table: t1
         type: ALL
possible_keys: title
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1396980
        Extra: Using temporary
*************** 2. row ***************
           id: 1
  select_type: SIMPLE
        table: t2
         type: ref
possible_keys: title
          key: title
      key_len: 77
          ref: imdb.t1.title
         rows: 1
        Extra: Using index
condition; Using where; Distinct
2 rows in set (0.00 sec)
```

Can you think of a more efficient way of doing the same?

ref_or_null

- **ref_or_null**
  - This join type is like *ref*, but with the addition that MySQL does an extra search for rows that contain `NULL` values.
  - This join type optimization is used most often in resolving subqueries.

ref_or_null (cont.)

```
mysql> EXPLAIN SELECT * FROM cast_info
       WHERE nr_order = 1 or nr_order IS NULL\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: ref_or_null
possible_keys: nr_order
          key: nr_order
      key_len: 5
          ref: const
         rows: 12193688
        Extra: Using index condition
1 row in set (0.00 sec)
```

index_merge

- Results from more than one index are combined either by *intersection* or *union*.
- In this case, the *key* column contains a list of indexes.

```
mysql> EXPLAIN SELECT * FROM title
WHERE title = 'Dracula' or production_year = 1922\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: index_merge
possible_keys: production_year,title
          key: title,production_year
      key_len: 77,5
          ref: NULL
         rows: 2895
        Extra: Using sort_union(title,production_year); Using where
1 row in set (0.00 sec)
```

unique_subquery/index_subquery

- **unique_subquery**
  - The result of a subquery is covered by a unique index.
  - The subquery is used within an `IN(...)` predicate.
- **index_subquery**
  - Similar to **unique_subquery**, only allowing for non-unique indexes

unique_subquery/index_subquery (cont.)

```
mysql> EXPLAIN SELECT * FROM title
       WHERE title = 'Bambi' and kind_id NOT IN
       (SELECT id FROM kind_type WHERE kind like 'tv%')\G
```

```
*************** 1. row ***************
           id: 1
  select_type: PRIMARY
        table: title
         type: ref
possible_keys: title
          key: title
      key_len: 77
          ref: const
         rows: 4
        Extra: Using where
```

```
*************** 2. row ***************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: kind_type
         type: unique_subquery
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: func
         rows: 1
        Extra: Using where
2 rows in set (0.04 sec)
```

For `index_subquery`, use a non-PRIMARY, non-UNIQUE key

range

- Only rows that are **in a given range** will be retrieved
- An **index** will still be **used** to select the rows
- The *key_len* contains the **longest key part** that is used
- The **ref** column will be `NULL` for this type

range (cont.)

```
mysql> EXPLAIN SELECT * FROM title
       WHERE title='Bambi' OR title='Dumbo' OR title='Cinderella'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: range
possible_keys: title
          key: title
      key_len: 77
          ref: NULL
         rows: 49
        Extra: Using where
1 row in set (0.00 sec)
```

index

- The **whole index tree** is scanned
- Otherwise same as **ALL**
- Faster than **ALL** since the index file is (should be) smaller than the data file
- MySQL can use this join type when the query uses only columns that are part of a single index

index (cont.)

```
mysql> EXPLAIN SELECT count(*), production_year,
          group_concat(DISTINCT kind_id ORDER BY kind_id) as kind_id
       FROM title
       GROUP BY production_year ORDER BY production_year\G
 *************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: index
possible_keys: NULL
          key: production_year
      key_len: 5
          ref: NULL
         rows: 1396980
        Extra: NULL
1 row in set (0.00 sec)
```

ALL

- A full table scan; **the entire table** is scanned
- Not good even for the first (non-const) table
- **Very bad** for subsequent tables, since it means a full table scan for each combination of rows from the previous tables is performed
- Solutions: **rephrase query**, add **more indexes**

ALL (cont.)

```
mysql> EXPLAIN SELECT * from title
WHERE MAKEDATE(production_year, 1) >= now() - INTERVAL 3 YEAR\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1442263
        Extra: Using where
1 row in set (0.00 sec)
```

What You Would Like to See

- **Using index**
  - Excellent! MySQL can search for the rows **directly from the index tree**, without reading the actual table (covering index)
- **Using where**
  - Good! If this is **missing**, you will get **all rows** from the table
- **Distinct**
  - Good! Only **one row for each combination** from the previous tables
- **Not exists**
  - Good! MySQL is able to do a `LEFT JOIN` optimization, and some rows can be left out

What You Don't Like to See

- **Using filesort**
  - Extra **sorting** pass needed!
- **Using temporary**
  - Temporary table needed!
  - Typically happens with different `ORDER BY` and `GROUP BY`
- **Using join buffer**
  - Tables are processed in large batches of rows, instead of by indexed lookups.
- **Range checked for each record (index map: N)**
  - Individual records are separately optimized for index retrieval
  - This is not fast, but faster than a join with no index at all.

Query Optimization

COMPOSITE INDEXES

Exercise: Add Index(es) to Improve this Query

3.41s

```
mysql> ALTER TABLE title DROP INDEX title;
mysql> EXPLAIN SELECT * FROM title WHERE title = 'Pilot' AND
production_year BETWEEN 1997 and 2009\G
************************** 1. row **************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1569823
        Extra: Using where
1 row in set (0.00 sec)
```

This number of rows is a guess. It keeps changing between examples.

* Note: *indexes* is the appropriate plural for an index in a database. Use *indices* for the stock market. Never use the back-formation *indice*.

We're Spoiled for Choice.

```
# Which one is best?
# ALTER TABLE title ADD INDEX py (production_year);
# ALTER TABLE title ADD INDEX t (title);
# ALTER TABLE title ADD INDEX py_t (production_year, title);
# ALTER TABLE title ADD INDEX t_py (title, production_year);

# Start by trying the production_year example:
mysql> ALTER TABLE title ADD INDEX py (production_year);
Query OK, 1543719 rows affected (38.07 sec)
Records: 1543719  Duplicates: 0  Warnings: 0
```

Index on Production_Year

3.53s

```
mysql> EXPLAIN SELECT * from title WHERE title = 'Pilot'
AND production_year BETWEEN 1997 and 2009\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ALL
possible_keys: py
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1592559
        Extra: Using where
1 row in set (0.02 sec)
```

How about a Smaller Range?

0.92s

```
mysql> EXPLAIN SELECT * from title WHERE title = 'Pilot'
AND production_year BETWEEN 2006 and 2009\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: range
possible_keys: py
          key: py
      key_len: 5
          ref: NULL
         rows: 148320
        Extra: Using where
1 row in set (0.00 sec)
```


0.02s

Index on Title

```
mysql> ALTER TABLE title ADD INDEX t (title(50));
Query OK, 1543719 rows affected (38.07 sec)
Records: 1543719  Duplicates: 0  Warnings: 0
mysql> EXPLAIN SELECT * from title WHERE title = 'Pilot'
AND production_year BETWEEN 2006 and 2009\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ref
possible_keys: py,t
          key: t
      key_len: 152
          ref: const
         rows: 926
        Extra: Using where
1 row in set (0.00 sec)
```

Comparing the Two

```
mysql> EXPLAIN SELECT * from title WHERE title =
'Pilot' AND production_year BETWEEN 2006 and 2009\G
```

```
           id: 1
  select_type: SIMPLE
        table: title
         type: range
possible_keys: py
          key: py
      key_len: 5
          ref: NULL
         rows: 148320
        Extra: Using where
1 row in set (0.00 sec)
```

```
           id: 1
  select_type: SIMPLE
        table: title
         type: ref
possible_keys: py,t
          key: t
      key_len: 152
          ref: const
         rows: 926
        Extra: Using where
1 row in set (0.00 sec)
```

Composite Indexes

- Which is better?
  - `INDEX py_t (production_year, title)`
  - `INDEX t_py (title, production_year)`

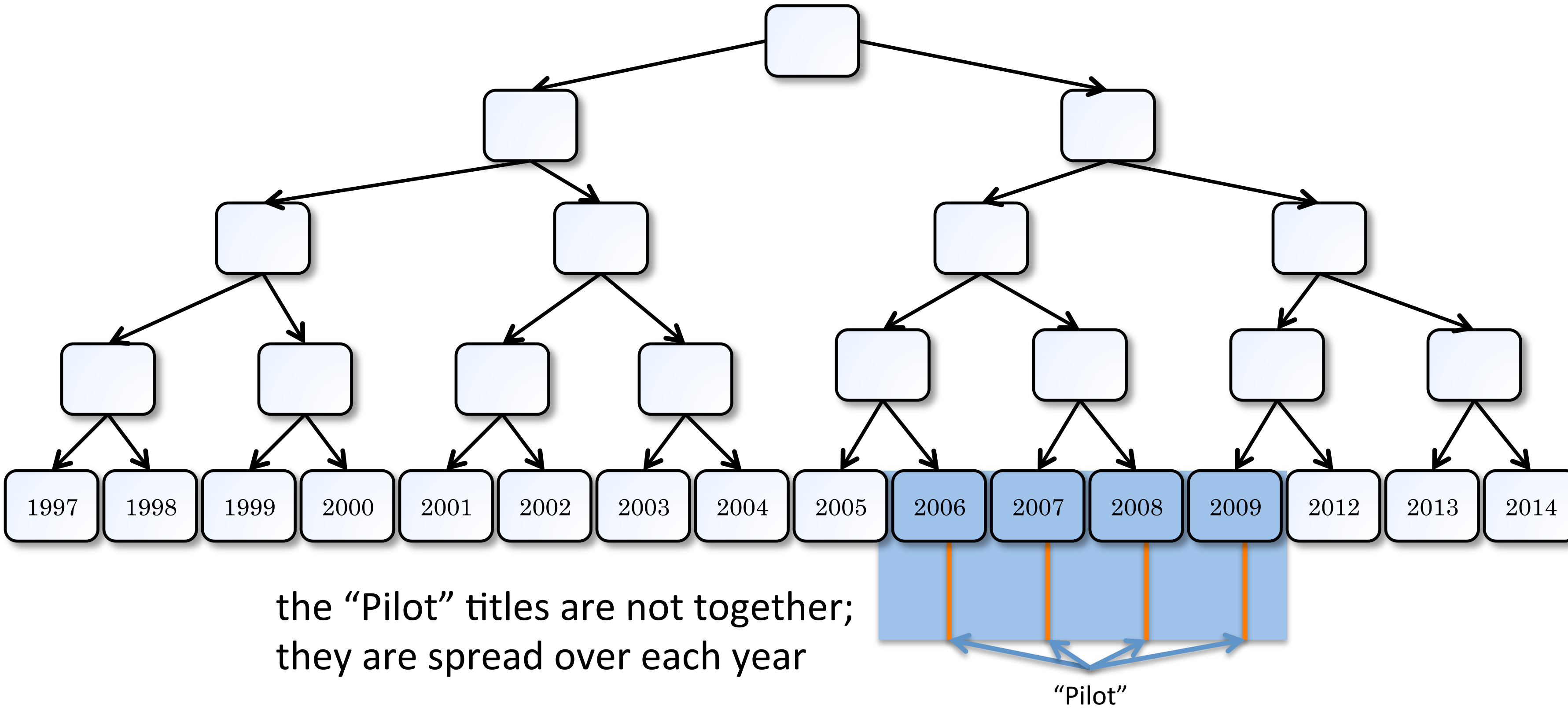
0.02s

Index on py_t

```
mysql> ALTER TABLE title ADD INDEX py_t
(production_year, title(50));
Query OK, 1543719 rows affected (2 min 15.64 sec)
Records: 1543719  Duplicates: 0  Warnings: 0
mysql> EXPLAIN SELECT * from title WHERE title = 'Pilot'
AND production_year BETWEEN 2006 and 2009\G
************************** 1. row **************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ref
possible_keys: py,t,py_t
          key: t
      key_len: 152
          ref: const
         rows: 926
        Extra: Using where
1 row in set (0.03 sec)
```

http://www.mysqlperformanceblog.com/2010/01/09/getting-around-optimizer-limitations-with-an-in-list/

Index on py_t



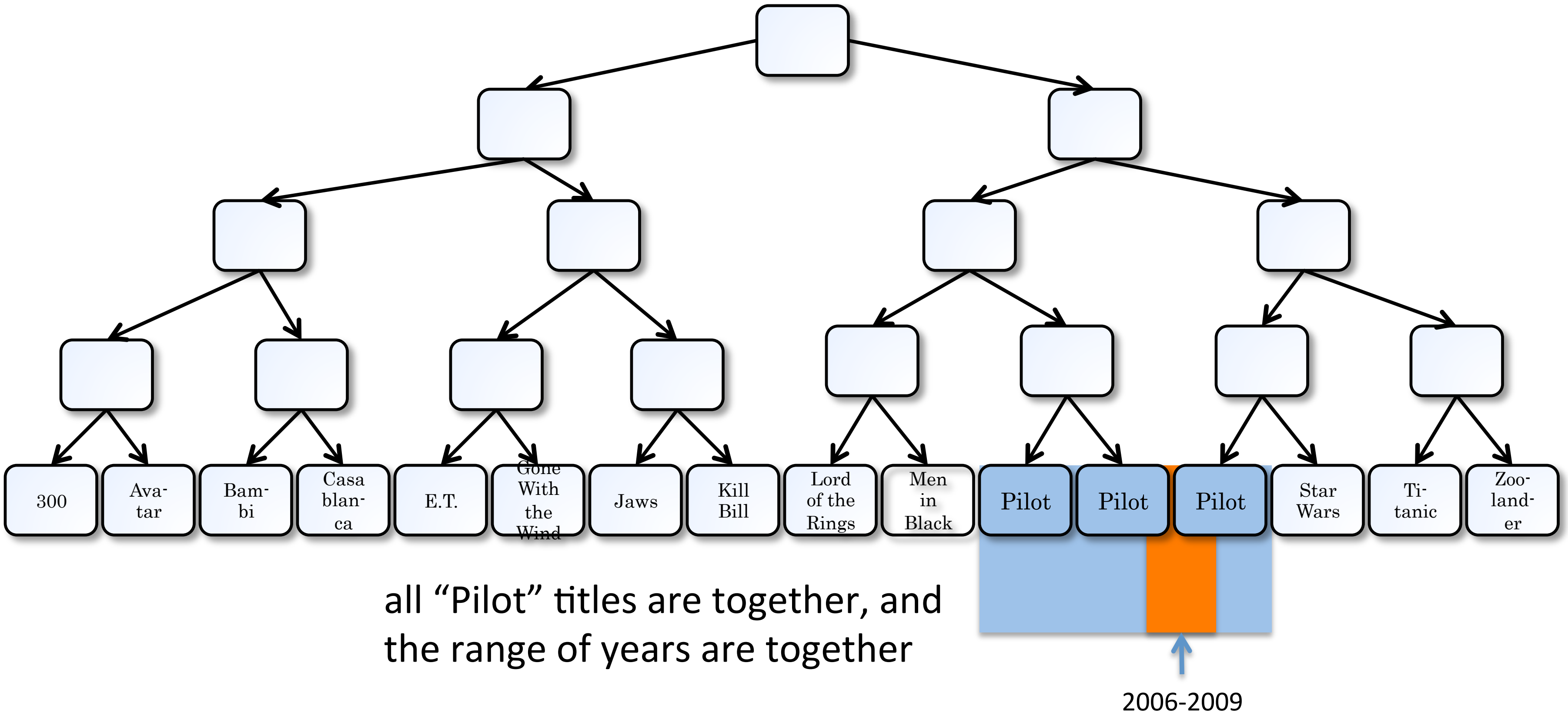
the "Pilot" titles are not together;
they are spread over each year

0.00s

Index on t_py

```
mysql> ALTER TABLE title ADD INDEX t_py
(title(50), production_year);
Query OK, 1543719 rows affected (1 min 52.63 sec)
Records: 1543719  Duplicates: 0  Warnings: 0
mysql> EXPLAIN SELECT * from title WHERE title = 'Pilot'
AND production_year BETWEEN 2006 and 2009\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: range
possible_keys: py,t,py_t,t_py
          key: t_py
      key_len: 157
          ref: NULL
         rows: 82
        Extra: Using where
1 row in set (0.08 sec)
```

Index on t_py



all “Pilot” titles are together, and the range of years are together

Recommendations

- Don't know what order to specify the columns?
  - RULE: Think about how the equality comparisons narrow down the subset of rows to examine. Define the index so the *leftmost* columns filter most effectively.
  - EXCEPTION: If you have a range comparison (`!=`, `<`, `>`, `BETWEEN`, `LIKE`), those columns should go to the *right* in the index.

http://www.mysqlperformanceblog.com/2010/01/09/getting-around-optimizer-limitations-with-an-in-list/

Recommendations (cont.)

- Columns after the range-comparison column can't be used for filtering in MySQL <5.6
  - but may still be useful in the index, as we'll see
- We can still push down those extra columns to the engine (>= 5.6), having a speed up if the condition is very selective

http://www.mysqlperformanceblog.com/2012/03/12/index-condition-pushdown-in-mysql-5-6-and-mariadb-5-5-and-its-performance-impact/

Using index condition (5.6)

```
mysql> EXPLAIN SELECT * FROM title
       WHERE title like 'B%' and
             production_year BETWEEN 1945 and 1950\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: range
possible_keys: title,production_year,production_year_title
          key: production_year_title
      key_len: 82
          ref: NULL
         rows: 23496
        Extra: Using index condition; Using where
1 row in set (0.00 sec)
```

Query Optimization

OTHER INDEXING TECHNIQUES

Indexes Are Multi-Purpose

- So far indexes have only been used for filtering.
  - This is the most typical case—**don't forget it**.
- There are also other ways MySQL can use indexes:
  - Avoiding having to sort.
  - Preventing temporary tables.
  - Avoiding reading rows from the tables.

The First Example Again

3.13s

```
mysql> EXPLAIN SELECT id,title,production_year FROM title
WHERE title  = 'Bambi' ORDER BY production_year\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1900087
        Extra: Using where; Using filesort;
1 row in set (0.00 sec)
```

Index Prevents Sort

0.00s

```
mysql> ALTER TABLE title ADD INDEX t_py
  (title(50), production_year);

mysql> EXPLAIN SELECT id,title,production_year FROM title
WHERE title  = 'Bambi' ORDER BY production_year\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ref
possible_keys: t_py
          key: t_py
      key_len: 152
          ref: const
         rows: 4
        Extra: Using where
1 row in set (0.00 sec)
```

Temporary Table

3.10s

```
mysql> EXPLAIN select count(*) as c, production_year FROM title
 GROUP BY production_year\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1524577
        Extra: Using temporary; Using filesort
1 row in set (0.00 sec)
```

Full Index Scan

0.00s

```
mysql> EXPLAIN select count(*) as c, production_year FROM title
 GROUP BY production_year\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: index
possible_keys: NULL
          key: py
      key_len: 5
          ref: NULL
         rows: 1524577
        Extra: Using index
1 row in set (0.00 sec)
```

ALTER TABLE title ADD INDEX py (production_year);

Retrieving Only Limited Columns

- Query:

```
SELECT person_id
FROM cast_info
WHERE person_role_id = 35721;
```

- What's the difference between indexes `(person_role_id)` versus `(person_role_id, person_id)`?

Retrieving Only Limited Columns

```
mysql> EXPLAIN SELECT person_id FROM cast_info
    WHERE person_role_id = 35721\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: ref
possible_keys: person_role_id
          key: person_role_id
      key_len: 5
          ref: const
         rows: 146
        Extra: Using where
1 row in set (0.01 sec)
```

ALTER TABLE cast_info ADD INDEX (person_role_id);

Covering Index Optimization

```
mysql> EXPLAIN SELECT person_id FROM cast_info
WHERE person_role_id = 35721\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: ref
possible_keys: person_role_id,person_role_id_person_id
          key: person_role_id_person_id
      key_len: 5
          ref: const
         rows: 146
        Extra: Using where; Using index
1 row in set (0.00 sec)
```

ALTER TABLE cast_info ADD INDEX person_role_id_person_id (person_role_id, person_id);

Prefix Index

- The problem with this schema, is there's just a couple of outliers with really long names:

```
mysql> SELECT max(length(title)) from title;
+--------------------+
| max(length(title)) |
+--------------------+
|                334 |
+--------------------+
1 row in set (6.64 sec)
mysql> SELECT max(length(name)) from char_name;
+-------------------+
| max(length(name)) |
+-------------------+
|               478 |
+-------------------+
1 row in set (7.80 sec)
```

Two Ways to Solve This

- Pick a good length to get a lot of uniqueness:

```
mysql> SELECT count(distinct(title)) as n_unique,
count(distinct(LEFT(title, 100))) as n100, count(distinct(LEFT(title, 75))) as n75,
count(distinct(LEFT(title, 50))) as n50, count(distinct(LEFT(title, 40))) as n40,
count(distinct(LEFT(title, 30))) as n30, count(distinct(LEFT(title, 20))) as n20,
count(distinct(LEFT(title, 10))) as n10  FROM title;
+----------+--------+--------+--------+--------+--------+--------+--------+
| n_unique | n100   | n75    | n50    | n40    | n30    | n20    | n10    |
+----------+--------+--------+--------+--------+--------+--------+--------+
|   998335 | 998320 | 998291 | 997887 | 996727 | 991532 | 960894 | 624949 |
+----------+--------+--------+--------+--------+--------+--------+--------+
```

96% uniqueness, but only 20 chars instead of 300+ Looks pretty good to me:ALTER TABLE title ADD index (name(20))

Option 2: Emulate a Hash Index

```
mysql> ALTER TABLE title ADD title_crc32 INT UNSIGNED, ADD INDEX (title_crc32);
mysql> UPDATE title SET title_crc32 = crc32(title);
mysql> SELECT count(distinct(BINARY title)),
count(distinct(title_crc32)) from title;
+-------------------------------+-------------------------------+
| count(distinct(BINARY title)) | count(distinct(title_crc32)) |
+-------------------------------+-------------------------------+
|                       1001509 |                      1001393 |
+-------------------------------+-------------------------------+
1 row in set (44.81 sec)
```

A good hashing algorithm has good distribution. How good is this?

Option 2: Hash Index (cont.)

- Query needs to be transformed slightly to:

```
SELECT * FROM title
WHERE title_crc32 = crc32('Bambi')
AND title = 'Bambi';
```

- All updates/inserts also need to update the value of `title_crc32` every time title changes.
  - Can be done easily via the application, or you can use a trigger if your write load low enough.

Pros & Cons of the Two Solutions

Prefix Index:

- **Pro:**
  Built in to MySQL/no magic required.
- **Cons:**
  Not very effective when the start of the string is not very unique.

Hash Index:

- **Pro:**
  Very Good when there is not much uniqueness until very far into the string.
- **Cons:**
  Equality searches only.
  Requires ugly magic to work with collations/ case sensitivity.

Index Hints

- Optimizer decision making is all about tradeoffs.
  - MySQL wants to pick the best plan, but it can't be exhaustive in deciding if it takes too long.
- If MySQL doesn't pick correctly, you can override:
  - `USE INDEX`
  - `FORCE INDEX`
  - `IGNORE INDEX`
- See: http://dev.mysql.com/doc/refman/5.6/en/index-hints.html

USE INDEX

- Tell the optimizer to consider *only* the named index.

```
mysql> SELECT * FROM title USE INDEX (py_t)
       WHERE production_year = 2009;
```

FORCE INDEX

- Like USE INDEX, consider only the named index.
- But also tells the optimizer that a table-scan is very expensive, so prefer to use the index, instead of analyzing the breakpoint when a table-scan may be easier.

```
mysql> SELECT * FROM title FORCE INDEX (title)
       WHERE title LIKE 'The %';
```

IGNORE INDEX

- Tells the optimizer *not* to use a specified index.

```
mysql> SELECT * FROM title IGNORE INDEX (t_p)
       WHERE title LIKE 'The %';
```

Caveats of Optimizer Hints

- Even USE INDEX or FORCE INDEX doesn't make an index help if it's totally inapplicable to the query:

```
mysql> SELECT * FROM title USE INDEX (t_py)
     WHERE production_year = 2009;
```

- The optimizer handles most cases well. If your query is costly, it's more likely that you have the wrong indexes than the optimizer is making a mistake.
- Hard-coding your application to use a specific index means that after you create the right index, you'll have to change your code as well.

Query Tuning Exercise

- It's now your turn to optimize these queries before we continue on:
  - ```
    SELECT * FROM name
    WHERE name = 'Brosnan, Pierce';
    ```
  - ```
    SELECT count(*) c, person_id
    FROM person_info
    GROUP by person_id;
    ```

Query Optimization

JOIN OPTIMIZATION

Join Analysis

```
mysql> EXPLAIN SELECT person_info.* FROM name INNER JOIN person_info
ON (name.id=person_info.person_id) AND name.name='Bana, Eric'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: name
         type: ref
possible_keys: PRIMARY,name
          key: name
      key_len: 152
          ref: const
         rows: 1
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: person_info
         type: ref
possible_keys: person_id
          key: person_id
      key_len: 4
          ref: imdb.name.id
         rows: 2
        Extra:
2 rows in set (0.01 sec)
```

```
mysql> EXPLAIN SELECT name.* FROM name INNER JOIN cast_info
ON name.id=cast_info.person_id INNER JOIN char_name
ON cast_info.person_role_id=char_name.id WHERE char_name.name = 'James Bond'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 22743540
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: name
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: imdb.cast_info.person_id
         rows: 1
        Extra:
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: char_name
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: imdb.cast_info.person_role_id
         rows: 1
        Extra: Using where
3 rows in set (0.07 sec)
```

4m2s

The order you see these tables mentioned is the order MySQL has decided to join on.

First Index

```
mysql> ALTER TABLE char_name ADD index name_idx (name(50));
Query OK, 2406561 rows affected (1 min 56.10 sec)
Records: 2406561 Duplicates: 0 Warnings: 0
```

```
mysql> EXPLAIN SELECT name.* FROM name INNER JOIN cast_info
ON name.id=cast_info.person_id INNER JOIN char_name
ON cast_info.person_role_id=char_name.id WHERE char_name.name = 'James Bond'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: char_name
         type: ref
possible_keys: PRIMARY,name_idx
          key: name_idx
      key_len: 152
          ref: const
         rows: 1
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 22743540
        Extra: Using where; Using join buffer
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: name
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: imdb.cast_info.person_id
         rows: 1
        Extra:
3 rows in set (0.24 sec)
```

1m48s

The order changed. cast_info was previously first!

```
mysql> EXPLAIN SELECT name.* FROM name INNER JOIN cast_info
ON name.id=cast_info.person_id INNER JOIN char_name
ON cast_info.person_role_id=char_name.id WHERE char_name.name = 'James Bond'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: char_name
         type: ref
possible_keys: PRIMARY,name_idx
          key: name_idx
      key_len: 152
          ref: const
         rows: 1
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: ref
possible_keys: person_role_id_person_id
          key: person_role_id_person_id
      key_len: 5
          ref: imdb.char_name.id
         rows: 4
        Extra: Using where; Using index
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: name
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: imdb.cast_info.person_id
         rows: 1
        Extra:
3 rows in set (0.17 sec)
```

0.00s

TIP: Using a covering index means that we retrieve all data directly from the index.

Join Methods

- You need to design queries and indexes to filter as fast as possible.
- MySQL main join method: a **nested loop** join.
- Alternative methods:
  - Batched key access (nested loop join optimized to avoid random disk access) –only in 5.6, limited usage
  - Hash joins –only for equijoins, only in MariaDB

Performance comparison: http://www.mysqlperformanceblog.com/2012/05/31/a-case-for-mariadbs-hash-joins/

Nested Loop Join Example

- Find all actors that were active between 1960 and 1970:

Actors:

id	first_name	last_name
1	Sean	Connery
2	George	Lazenby
3	Roger	Moore
4	Timothy	Dalton
5	Pierce	Brosnan
6	Daniel	Craig

Movies:

id	name	year
1	Dr. No	1962
2	From Russia with Love	1963
3	Goldfinger	1964
4	You only live twice	1967
5	On Her Majesty's Secret Service	1969
6	Diamonds Are Forever	1971

Nested Loop Join Example (cont.)

- Find all actors that were active between 1960 and 1970:

Actors:

id	first_name	last_name
1	Sean	Connery
2	George	Lazenby
3	Roger	Moore
4	Timothy	Dalton
5	Pierce	Brosnan
6	Daniel	Craig

Movies:

id	name	year
1	Dr. No	1962
2	From Russia with Love	1963
3	Goldfinger	1964
4	You only live twice	1967
5	On Her Majesty's Secret Service	1969
6	Diamonds Are Forever	1971

If That Query Is Common

- When you can't filter enough on one table, bring some of the other filters from the other tables to the first one:

Actors:

id	first_name	last_name	start_date	finish_date
1	Sean	Connery	1962	1971
2	George	Lazenby	1969	1969
3	Roger	Moore	1973	1985
4	Timothy	Dalton	1987	1989
5	Pierce	Brosnan	1995	2002
6	Daniel	Craig	2006	2011

STRAIGHT_JOIN

- Tells the optimizer not to reorder tables; access tables in exactly the order you gave in the query.
- Use it like a query modifier like DISTINCT:

```
mysql> SELECT STRAIGHT_JOIN name.*
       FROM char_name
       INNER JOIN cast_info
         ON name.id = cast_info.person_role_id
       INNER JOIN name
         ON cast_info.person_id = name.id
         WHERE char_name.name = 'James Bond';
```

Things Are Looking Good!?

- Please ***don't*** take away that adding indexes is the only secret to performance.
- There's more to consider:
  - Optimizer limitations for subqueries
  - Estimating index prefix length
  - Join methods
  - Optimizer hints
  - Advanced query profiling

Query Optimization

SUBQUERY OPTIMIZATION

Subquery Analysis

4.84s

```
mysql> EXPLAIN SELECT * FROM title WHERE kind_id IN
(SELECT id FROM kind_type WHERE kind='video game')\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: title
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1567676
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: kind_type
         type: const
possible_keys: PRIMARY,kind_id
          key: kind_id
      key_len: 47
          ref: const
         rows: 1
        Extra: Using index
2 rows in set (0.00 sec)
```

Will it fix it if we add an index on title.kind_id?

With Index on `kind_id`

4.9s

```
mysql> EXPLAIN SELECT * FROM title WHERE kind_id IN
(SELECT id FROM kind_type WHERE kind='video game')\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: title
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1574389
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: kind_type
         type: const
possible_keys: PRIMARY,kind_id
          key: kind_id
      key_len: 47
          ref: const
         rows: 1
        Extra: Using index
2 rows in set (0.11 sec)
```

No! It doesn't.
Why is this?

0.07s

Scalar Subquery

```
mysql> EXPLAIN SELECT * FROM title WHERE kind_id =
(SELECT id FROM kind_type WHERE kind='video game')\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: title
         type: ref
possible_keys: k
          key: k
      key_len: 4
          ref: const
         rows: 8502
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: SUBQUERY
        table: kind_type
         type: const
possible_keys: kind_id
          key: kind_id
      key_len: 47
          ref:
         rows: 1
        Extra: Using index
2 rows in set (0.10 sec)
```

Change to using equality, it works!

The optimizer treats *scalar* subqueries differently, but only checks that the subquery is scalar if you use =.

0.06s

Solving via Join

```
mysql> EXPLAIN SELECT title.* FROM title INNER JOIN kind_type ON
(title.kind_id = kind_type.id) WHERE kind_type.kind IN ('video game')\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: kind_type
         type: const
possible_keys: PRIMARY,kind_id
          key: kind_id
      key_len: 47
          ref: const
         rows: 1
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: ref
possible_keys: kind_id
          key: kind_id
      key_len: 4
          ref: const
         rows: 8502
        Extra:
2 rows in set (0.00 sec)
```

It's okay to have multiple kind's specified using this syntax.

Query Optimization

BEYOND EXPLAIN

The Limitations of EXPLAIN

- EXPLAIN shows MySQL's intentions; there is no post-execution analysis.
  - How many rows actually had to be sorted?
  - Was that temporary table created on disk?
  - Did the LIMIT 10 result in a quick match, resulting in fewer rows scanned?
  - … we don't know.

Going PRO++

- Combine EXPLAIN with other MySQL diagnostics:
  - `SHOW SESSION STATUS`
  - `SHOW PROFILES`
  - Slow Query Log Extended Statistics in Percona Server

Why Going PRO Is Important

- In addition to the limitations on the previous slide, MySQL occasionally introduces new undocumented features.
  - When preparing this presentation, we found BUG #50394!
  - http://bugs.mysql.com/bug.php?id=50394

```
mysql-5141> EXPLAIN select STRAIGHT_JOIN count(*) as c, person_id
FROM cast_info FORCE INDEX(person_id) INNER JOIN title ON
(cast_info.movie_id=title.id) WHERE title.kind_id = 1
GROUP BY cast_info.person_id ORDER by c DESC LIMIT 1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: cast_info
         type: index
possible_keys: NULL
          key: person_id
      key_len: 8
          ref: NULL
         rows: 8
        Extra: Using index; Using temporary; Using filesort
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: title
         type: eq_ref
possible_keys: PRIMARY,title_kind_id_exists
          key: PRIMARY
      key_len: 4
          ref: imdb.cast_info.movie_id
         rows: 1
        Extra: Using where
2 rows in set (0.00 sec)
```

16m

Find the actor that starred in the most movies.

MySQL says that only 8 rows were examined in 5.1.41

Double Checking

```
mysql> show status like 'ha%';
+----------------------------+----------+
| Variable_name              | Value    |
+----------------------------+----------+
| Handler_commit             | 0        |
| Handler_delete             | 0        |
| Handler_discover           | 0        |
| Handler_prepare            | 0        |
| Handler_read_first         | 1        |
| Handler_read_key           | 13890229 |
| Handler_read_next          | 14286456 |
| Handler_read_prev          | 0        |
| Handler_read_rnd           | 0        |
| Handler_read_rnd_next      | 2407004  |
| Handler_rollback           | 0        |
| Handler_savepoint          | 0        |
| Handler_savepoint_rollback | 0        |
| Handler_update             | 0        |
| Handler_write              | 2407001  |
+----------------------------+----------+
15 rows in set (0.00 sec)
```

http://dev.mysql.com/doc/refman/5.1/en/server-status-variables.html

"The number of times the first entry in an index was read"

"The number of requests to read a row based on a key."

"The number of requests to read the next row in key order."

"The number of requests to read the next row in the data file."

"The number of requests to insert a row in a table."

SHOW PROFILES

Enable profiling:

```
mysql> SET profiling = 1;
```

Run some query(s):

```
mysql> SELECT STRAIGHT_JOIN COUNT(*) AS c, person_id
FROM cast_info FORCE INDEX(person_id)
INNER JOIN title ON  (cast_info.movie_id=title.id)
WHERE title.kind_id = 1
GROUP BY cast_info.person_id
ORDER by c DESC LIMIT 1;
```

In seconds

Only shows queries from your current session.

View the report:

```
mysql> SHOW PROFILES;
| Query_ID | Duration     | Query
|        1 | 211.21064300 | SELECT STRAIGHT_JOIN ...
```


SHOW PROFILES (cont.)

This was executed on a machine with an SSD drive (different timing)

```
mysql> show profile for query 1;
+------------------------------+-----------+ ..
| Status                       | Duration  | | Copying to tmp table         | 113.862209 |
+------------------------------+-----------+ | converting HEAP to MyISAM    |   0.200272 |
| starting                     |  0.002133 | | Copying to tmp table on disk |  96.506704 |
| checking permissions         |  0.000009 | | Sorting result               |   0.634087 |
| checking permissions         |  0.000009 | | Sending data                 |   0.000047 |
| Opening tables               |  0.000035 | | end                          |   0.000006 |
| System lock                  |  0.000022 | | removing tmp table           |   0.004839 |
| init                         |  0.000033 | | end                          |   0.000016 |
| optimizing                   |  0.000020 | | query end                    |   0.000004 |
| statistics                   |  0.000032 | | freeing items                |   0.000064 |
| preparing                    |  0.000031 | | logging slow query           |   0.000004 |
| Creating tmp table           |  0.000032 | | logging slow query           |   0.000003 |
| Sorting for group            |  0.000021 | | cleaning up                  |   0.000006 |
| executing                    |  0.000005 | +------------------------------+------------+
..                                             25 rows in set (0.00 sec)
```

Verbose Slow Log in Percona Server

```
SET GLOBAL long_query_time = 0;
SET GLOBAL log_slow_verbosity = 'full'; /* Percona Server */

# Time: 100924 13:58:47
# User@Host: root[root] @ localhost []
# Thread_id: 10  Schema: imdb  Last_errno: 0  Killed: 0
# Query_time: 399.563977  Lock_time: 0.000110  Rows_sent: 1  Rows_examined: 46313608
Rows_affected: 0  Rows_read: 1
# Bytes_sent: 131  Tmp_tables: 1  Tmp_disk_tables: 1  Tmp_table_sizes: 25194923
# InnoDB_trx_id: 1403
# QC_Hit: No  Full_scan: Yes  Full_join: No  Tmp_table: Yes  Tmp_table_on_disk: Yes
# Filesort: Yes  Filesort_on_disk: Yes  Merge_passes: 5
#   InnoDB_IO_r_ops: 1064749  InnoDB_IO_r_bytes: 17444847616  InnoDB_IO_r_wait:
26.935662
#   InnoDB_rec_lock_wait: 0.000000  InnoDB_queue_wait: 0.000000
#   InnoDB_pages_distinct: 65329
SET timestamp=1285336727;
select STRAIGHT_JOIN count(*) as c, person_id  FROM cast_info FORCE INDEX(person_id)
INNER JOIN title ON  (cast_info.movie_id=title.id) WHERE title.kind_id = 1  GROUP BY
cast_info.person_id ORDER by c DESC LIMIT 1;
```

This was executed on a machine with entirely cold caches.

Query Planning Overhead

- Some queries might spend too much time on the query planning and optimization phase
  - There is no query plan cache in MySQL
- It can be easily identified when `EXPLAIN` is “slow”
- Solution:
  - Force the plan with `STRAIGHT_JOIN`, etc.
  - Reduce `optimizer_search_depth` variable

Index Merge Optimization

- Index Merge access type allows the usage of more than one index per table access
- Types:
  - union of several conditions
    ```
    SELECT * FROM title where title = 'Bambi'
    or production_year = 1927
    ```
  - intersection
    ```
    SELECT * FROM title where title = 'Bambi'
    and production_year = 1927
    ```

The Merge Access Problem

- In both cases, there is usually faster alternative query plans:
  - Union: UNION clauses, single index usage, ...
  - Intersection: **composite indexes**, secondary index extensions, single index usage, ...
- MySQL merge algorithm is selected in most cases even if those other methods were available and slower
  - MySQL 5.6 fixes this, although not in all cases

Intersection Merge

- Drop the single-column indexes (if not used for other queries) and create a composite index with all columns
  - Even if the condition cannot be applied, a single column index or Index Condition Pushdown will be probably faster
- You can disable the merge intersection with:

```
SET optimizer_switch =
'index_merge_intersection=OFF'
```

http://www.mysqlperformanceblog.com/2012/12/14/the-optimization-that-often-isnt-index-merge-intersection/

Union Merge

- Union Merge may or may not be faster than other methods
  - In 5.6, ref or range over composite indexes should have higher preference
- The alternative would be converting the clause to a UNION/UNION ALL
  - UNION is not always better as it has a problem: for most cases it will create a temporary table on disk

Example Union Merge

```
SELECT id FROM title WHERE (title ='Pilot'
or episode_nr = 1) and production_year >
1977;
```

is slower than the equivalent:

```
SELECT id FROM title WHERE (title ='Pilot'
and production_year > 1977)
UNION
SELECT id FROM title WHERE (episode_nr = 1
and production_year > 1977);
```

More Features & Workarounds

- “Delayed Join”
  - http://www.mysqlperformanceblog.com/2007/04/06/using-delayed-join-to-optimize-count-and-limit-queries/
- The IN() list workaround
  - http://www.mysqlperformanceblog.com/2010/01/09/getting-around-optimizer-limitations-with-an-in-list/

Additional Exercises

```
– SELECT * FROM movie_info
  WHERE movie_id IN (SELECT id FROM title WHERE
  title = 'Batman Begins');
– SELECT * from title
  WHERE season_nr != 6 and title LIKE 'Best of%';
– SELECT t.title FROM name n
  INNER JOIN cast_info i ON n.id=i.person_id
  INNER JOIN title t ON i.movie_id=t.id
  INNER JOIN char_name c ON c.id=i.person_role_id
  WHERE n.name='Brosnan, Pierce' AND t.kind_id = 1
  AND  c.name='James Bond';
```

Schema Design

Percona Training

http://www.percona.com/training

Table of Contents

Why Is Schema Design Important?

- The schema is your collection of tables, indexes, constraints and other database objects.
- The key to good performance:
  - How you store data (the schema).
  - How you retrieve data (the queries).

What Makes a Good Schema?

- A well-designed schema stores all the data you need to store, and disallows invalid data.
- An "optimized" schema is really designed for the queries you need to run.
- If you need better performance, you can often improve it by making changes to the schema.

What Changes?

- SQL Data Types
- Table Design
- Normalization and Denormalization
- Index Design
- SQL Constraints
- Partitioning
- Data Warehousing
- Schema Design Tools
- Extensible Schema Design
- Views
- Triggers
- Stored Procedures and Functions

Understanding Requirements

- To design a schema, you need to know:
  - What data entities the project needs to store. *Movies, actors, users, ratings…*
  - How data entities are related. *Actors star in movies; TV episodes belong to a TV show; users give ratings to movies…*
- To optimize a schema, you need to know:
  - What queries the project needs to run.
  - Which queries are most important.

Data Definition Language (DDL)

- SQL commands to implement the schema:

```
CREATE TABLE name (
    …columns, indexes, constraints…
);
ALTER TABLE name
    …clauses…;
DROP TABLE name;
```

- A few other non-standard statements are DDL:

```
RENAME, TRUNCATE, etc.
```

Data Manipulation Language (DML)

- SQL statements that act on data:
  - CALL
  - DELETE
  - INSERT
  - LOAD
  - REPLACE
  - SELECT
  - UPDATE

What About Other Commands?

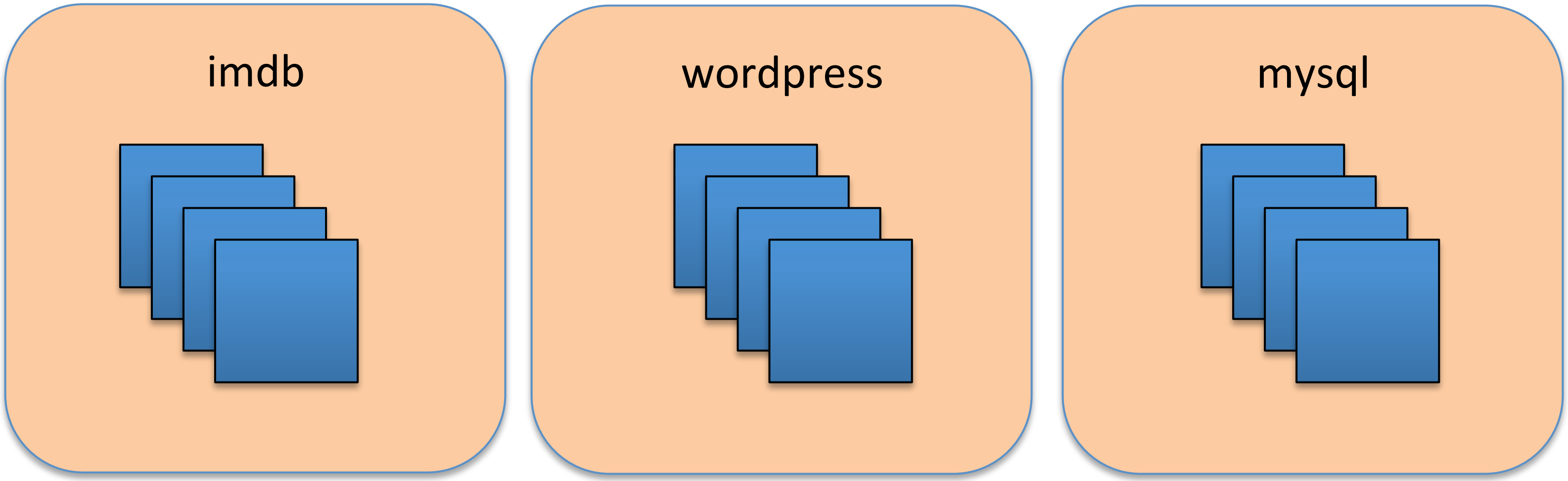
- There are other statements, but they count as neither DDL nor DML.
  - DESCRIBE
  - EXPLAIN
  - FLUSH
  - GRANT / REVOKE
  - SET
  - SHOW
  - etc.

Schema Design

DATABASE DESIGN

What Is a Database?

- *Database* and *schema* are synonyms in MySQL.
- A database is a “namespace” for tables.



PERCONA TRAINING

Uses of Databases

- Defining logical groups of tables and other objects.
- Allowing multiple tables to use the same name.
- Backing up groups of tables.
- Assigning privileges

Special Databases

- `mysql`
  - System tables for MySQL users, passwords, privileges, character sets, stored procedures, logs, etc.
- `information_schema`
  - Dynamic views into metadata and other configuration.
- `performance_schema`
  - Dynamic views into runtime status.
- `test*` (any database name that starts with 'test…')
  - Always readable by any user.

Common Database Elements

- Tables
- Indexes
- Constraints
- Views
- Stored Routines & Triggers

Schema Design

SQL DATA TYPES

Integer Types

- MySQL supports four sizes of integer:
  - TINYINT (1 byte) from -128 to 127
  - SMALLINT (2 bytes) from -32768 to 32767
  - INT (4 bytes) from -2147483648 to 2147483647
  - BIGINT (8 bytes) from -9223372036854775808 to 9223372036854775807
- The UNSIGNED option does not change the size or the number of distinct values, but allows only values >= 0.

Integer Types

Type	Bytes	Min Value	Max Value
TINYINT	1	-128	127
TINYINT UNSIGNED	1	0	255
SMALLINT	2	-32768	32767
SMALLINT UNSIGNED	2	0	65535
INT	4	-2147483648	2147483647
INT UNSIGNED	4	0	4294967295
BIGINT	8	-9223372036854775808	9223372036854775807
BIGINT UNSIGNED	8	0	18446744073709551615

Which Integer to Use?

- Choose the smallest type that supports the range of values you need to store.
  - `INT` vs. `BIGINT` is 4 bytes extra. Thiswhich adds up once you have millions of rows.
- Some columns may grow without bound (e.g. auto-increment primary keys), but others have a natural maximum value.
  - Number of players on a football team, for instance.

Does Size Matter?

- Of course!
  - Storing data more compactly means you can store more rows in the same space, both on disk and in memory.
  - If more of your data fits in memory, this benefits performance.

Integer Display Width

- Integer types have an optional argument, but this *does not* affect the size of the integer, nor the range of values it accepts—it's only a hint for the display width:

```
CREATE TABLE t (i INT(9) ZEROFILL);
INSERT INTO t VALUES (1234);
SELECT * FROM t;
+-----------+
| i         |
+-----------+
| 000001234 |
+-----------+
```

Float and Double

- FLOAT and DOUBLE are *inexact* numerics. They have necessary rounding behavior:

```
CREATE TABLE t (f FLOAT);
INSERT INTO t VALUES (3.3333333333);
SELECT f * 1000000000 FROM t;
+--------------------+
| f * 1000000000     |
+--------------------+
| 3333333253.8604736 |
+--------------------+
```

http://dev.mysql.com/doc/refman/5.6/en/problems-with-float.html

Why Is This a Problem?

- Don't use FLOAT or DOUBLE when you don't want rounding, or you need to search for a specific value.

```
SELECT * FROM t WHERE f = 3.3333333333;
Empty set (0.00 sec)
```

In Other Words...

If I had a dime for every time
I've seen someone use FLOAT to store currency,
I'd have $999.997634.
#ieee754jokes

Why Use Inexact Numbers?

- Use FLOAT or DOUBLE when you want to represent very small or very large values for measurements. For example, in scientific applications.

```
INSERT INTO t (f) VALUES (3e-12), (3e+12);
SELECT * FROM t;
+----------------+
| f              |
+----------------+
| 0.000000000003 |
|  3000000000000 |
+----------------+
```

Decimal

- DECIMAL (and its synonym NUMERIC) are *exact* scaled numerics—they don't round values.
- Use this for currency values.

```
CREATE TABLE t (d DECIMAL(9,2));
INSERT INTO t VALUES (3.33);
SELECT * FROM t;
+------+
| d    |
+------+
| 3.33 |
+------+
```

Decimal vs. Integer

- The size arguments of Decimal affect storage and range, unlike the width argument of Integer types.

```
DECIMAL(9,2)          INT(9)
```

Number of significant digits stored.

Scale (number of digits to the right of the decimal).

Number of digits displayed.

Date and Related Types

- Using date and time datatypes supports indexing—values in *YYYYMMDD* format are sorted in chronological order.
  - `DATE` (3 bytes) stores only *YYMMDD*
  - `TIME` (3 bytes) stores *HH:MM:SS*
  - `DATETIME` (8 bytes) stores *YYMMDD HH:MM:SS*
  - `TIMESTAMP` (4 bytes) also stores *YYMMDD HH:MM:SS*, converts value between system timezone and UTC.
  - `YEAR` (1 byte) stores only *YYYY*.

Choosing a Date or Time Type

- Use the most compact type that supports the values you need to store.
- For instance, if you don't need to store the time portion, using DATE makes your data more compact and makes some queries simpler.

```
WHERE datetime BETWEEN '2013-10-17 00:00:00'
AND '2013-10-17 23:59:59'
```

```
WHERE date = '2013-10-17'
```

Default Timestamps

```
CREATE TABLE Log (
    ts TIMESTAMP
        DEFAULT CURRENT_TIMESTAMP
        ON UPDATE CURRENT_TIMESTAMP
);
```

In MySQL 5.6+, this works for TIME and DATETIME columns too.

Searching Dates by Index

- You can use an index if your search uses the leftmost portion of the value.
- Wrong:

```
SELECT * FROM title WHERE
MONTH(production_year) = 4;
```

- Right:

```
SELECT * FROM title WHERE production_year
BETWEEN '2013-04-01' AND '2013-04-30';
```

Fractional Seconds

- DATETIME, TIME, and TIMESTAMP also have an optional precision for fractional seconds.

```
CREATE TABLE t (d DATETIME(6));
INSERT INTO t VALUES (NOW(6));
SELECT * FROM t;
+----------------------------+
| d                          |
+----------------------------+
| 2013-04-12 13:35:44.000000 |
+----------------------------+
```

Range 0-6, default 0.

Fractional time is a feature of MySQL 5.6

String Types

- Fixed-length strings: `CHAR(20)`
  - Storage always takes 20 characters.
- Variable-length strings: `VARCHAR(20)`
  - You can store a shorter string, and it takes only as many characters as the length of your string, plus 1 or 2 bytes for the length of the given string.
  - However, the string pads out to the maximum length when loaded from the storage engine into the SQL layer (e.g. result sets, sorting, temp tables).

Pads Out?!

- This is a good reason to define a VARCHAR length as compactly as possible.
- Antipattern: VARCHAR(255) as default string type.
  - Makes no sense for strings that can never be that long, e.g. postal codes, IP addresses, MD5 hashes.

Character Sets

- A property of character-based types
  - They limit the characters available
  - How they are encoded
  - How much spaces they take
- Examples:
  - utf8
  - ascii
  - latin1

Facts About Character Sets

- A database, a table and a row can have a defined character set, but the first two are only *defaults.*
- `utf8` in MySQL is a variable-length encoding between 1 and 3 bytes per character.
  - But strings pad out to 3 bytes per character in memory, sorting, temp tables, etc.
  - `utf8` only covers the Basic Multilingual Plane. For full UTF-8 (4 bytes) support, use `utf8mb4`.

Collation

- A collation is linked to a charset. It modifies the way character strings are compared by default (i.e.: case sensitiveness).
  - As a consequence, they also modify the sorting order
- Some example collations:
  - latin1_german2_ci
  - latin1_general_ci
  - latin1_general_cs
  - latin1_bin
  - latin1_spanish_ci

Character Set Pitfalls

- Comparing strings of different character sets or collations spoils the benefit of indexes, which relies on sorting order. Joins will be a lot slower.
- Mismatching the character sets for the client, the connection, and the database can cause confusing effects as text is converted back and forth.
- Storing text strings in a binary column loses character set information.

BLOB and TEXT

- BLOB for binary data of variable length.
- TEXT for text of variable length, with character set.

TINYBLOB	TINYTEXT	255	
BLOB	TEXT	65535	64KB - 1
MEDIUMBLOB	MEDIUMTEXT	16777215	16MB - 1
LONGBLOB	LONGTEXT	4294967295	4GB - 1

Q: What's the difference between VARCHAR(65535) and TEXT?
A: VARCHAR can have a DEFAULT value.

Indexing BLOB and TEXT

- Maximum index length is 1000 bytes*, so you must use a prefix index.

```
ALTER TABLE title ADD KEY (title(50));
```

prefix length

* Remember UTF-8 characters count as 3 bytes per character!

ENUM

- MySQL-specific data type (not standard SQL, and not supported by other RDBMS brands).
- You declare the data type as a list of string values, and MySQL stores these strings once, as part of the table definition.

```
CREATE TABLE t (sex ENUM('male', 'female'));
```

- When you insert a row, the value of the ENUM column is just an integer—the ordinal position of the string in the ENUM.

Sorting by ENUM

- Sort order may be counter-intuitive.

```
INSERT INTO t VALUES ('female'), ('male');
SELECT * FROM t ORDER BY sex;
+--------+
| sex    |
+--------+
| male   |
| female |
+--------+
```

sort order is by the ordinal integer, not alphabetical

SET

- Another MySQL-specific data type (not standard).
- Like `ENUM`, you declare a list of string values:

```
CREATE TABLE name (
    name TEXT,
    skills SET('actor', 'dancer', 'singer')
);
```

- When you insert a row, a single column can accept any combination of *multiple* values from the `SET`.

```
INSERT INTO name VALUES
(‘Zooey Deschanel’, ‘actor,singer’);
```

- The value is stored as a bitfield of up to 64 bits internally, so the `SET` may have up to 64 elements.

Drawbacks of ENUM and SET

- You need to ALTER TABLE to change the values.
- Reordering or deleting a value is a table restructure.
- Awkward to query the permitted values.

```
SELECT COLUMN_TYPE
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME='t';

+-----------------------+
| column_type           |
+-----------------------+
| enum('male','female') |
+-----------------------+
```

now I have to parse this

Mapping SQL Types to Java

- JDBC result sets return `java.lang.Object` for most SQL types.
  - The object is automatically given a more specific Java type, based on the SQL data type.
  - Except binary strings, which return `bytes[]`.

Mapping SQL Types to Java

```
ResultSet rs = stmt.executeQuery(
 "SELECT * FROM test.test_data_types");
ResultSetMetaData rsm = rs.getMetaData();
while (rs.next()) {
 for (i=1; i<=rsm.getColumnCount(); ++i) {
  Object o = rs.getObject()
  System.out.println(
   o.getClass().getName()+" "+o.toString()
  );
 }
}
```

Mapping SQL Types to Java

SQL Type	Java Type	Precision
BIGINT	java.lang.Long	20
BIGINT UNSIGNED	java.math.BigInteger	20
INT	java.lang.Integer	11
INT UNSIGNED	java.lang.Long	10
SMALLINT	java.lang.Integer	6
SMALLINT UNSIGNED	java.lang.Integer	5
TINYINT	java.lang.Integer	4
TINYINT UNSIGNED	java.lang.Integer	3

Mapping SQL Types to Java

SQL Type	Java Type	Precision
FLOAT	java.lang.Float	12
DOUBLE	java.lang.Double	22
DECIMAL(9,2)	java.lang.BigDecimal	9
BIT(1)	java.lang.Boolean	1

Mapping SQL Types to Java

SQL Type	Java Type	Precision
DATE	java.sql.Date	10
DATETIME	java.sql.Timestamp	19
DATETIME(6)	java.sql.Timestamp	26
TIME	java.sql.Time	10
TIME(6)	java.sql.Time	17
TIMESTAMP	java.sql.Timestamp	19
TIMESTAMP(6)	java.sql.Timestamp	26
YEAR	java.sql.Date	4

Mapping SQL Types to Java

SQL Type	Java Type	Precision
CHAR(*N*)	java.lang.String	*N*
VARCHAR(*N*)	java.lang.String	*N*
LONGTEXT	java.lang.String	2147483647
MEDIUMTEXT	java.lang.String	16777215
TEXT	java.lang.String	65535
TINYTEXT	java.lang.String	255
ENUM('A', 'B', 'C')	java.lang.String	1
SET('A', 'B', 'C')	java.lang.String	5

Mapping SQL Types to Java

SQL Type	Java Type	Precision
LONGBLOB	byte[]	2147483647
MEDIUMBLOB	byte[]	16777215
BLOB	byte[]	65535
TINYBLOB	byte[]	255
BINARY(*N*) or CHAR(*N*) CHARACTER SET BINARY	byte[]	*N*
VARBINARY(*N*) or VARCHAR(*N*) CHARACTER SET BINARY	byte[]	*N*
BIT(*N*) for *N* > 1	byte[]	*N*

`getObject()` returns garbage for these; use `getBytes()` instead

Encrypted Data

- MySQL doesn't support a transparently encrypted data type. Alternatives:
  - Store your data directory on an encrypted filesystem.
  - Encrypt in your application before inserting with SQL.
  - Encrypt/decrypt using a built-in MySQL functions:

```
INSERT INTO mytable (comment)
VALUES (AES_ENCRYPT('shhh!', 'password'));
```

```
SELECT AES_DECRYPT(comment, 'password')
FROM mytable;
```

Schema Design

TABLE DESIGN

What Is a Table?

- Tables have headings that define column names and data types.
- Tables have rows, that have the same columns as the heading.
- Each column has the same name and data type on every row of that table.

What Tables Do I Need?

- Each data entity in your application.
- Each attribute that may have multiple values.
- Lookup tables.
- Many-to-many relationship tables.
- Temporary tables.

Data Entity Tables

- One table for each distinct entity in your application
  - E.g. titles, users, keywords.
- Each entity has its own set of attribute columns.
  - If you have two objects with different attributes, it's a clue that they are two separate types of entities and need two separate tables.

Multi-Valued Attribute Tables

- If an attribute can have multiple values, make a new table.
  - E.g. movie info:

```
CREATE TABLE `movie_info` (
  id int AUTO_INCREMENT PRIMARY KEY,
  movie_id int NOT NULL,
  info_type_id int NOT NULL,
  info text NOT NULL,
  note text,
  FOREIGN KEY (movie_id) REFERENCES title(id)
);
```

  - Your entity tables reference a lookup table with a foreign key.

Lookup Tables

- An attribute may be limited to a finite set of values.
  - E.g. role_type
- Create a separate table to list the permitted values.

```
CREATE TABLE role_type (
  id int AUTO_INCREMENT PRIMARY KEY,
  role varchar(32) NOT NULL,
  UNIQUE KEY `role` (`role`)
) ENGINE=InnoDB AUTO_INCREMENT=13;
```

  - Lookup tables typically have a small number of rows, and don't get frequent updates.
  - Your entity tables reference a lookup table with a foreign key.

Many-to-Many Relationship Tables

- Many-to-many relationships should use an intersection table:
  - E.g. movies have many keywords, keywords can apply to many movies.

```
CREATE TABLE movie_keyword (
  movie_id INT NOT NULL,
  keyword_id INT NOT NULL,
  PRIMARY KEY (movie_id, keyword_id),
  FOREIGN KEY (movie_id) REFERENCES title(id),
  FOREIGN KEY (keyword_id) REFERENCES keyword(id)
);
```

  - These tables reference *both* entity tables with foreign keys.

Temporary Tables

- Some queries create temp tables *implicitly*.
  - `GROUP BY`, `ORDER BY`, `UNION`, views, subqueries. Sometimes you can avoid this with the right indexes.
  - Created as `MEMORY`, may spool to disk as `MyISAM`.
- You can also create a temp table *explicitly*.
  - `CREATE TEMPORARY TABLE …`
  - Good for interim result sets.
- Temp tables are visible only to the session, and are dropped automatically when the session ends.

One Column per Attribute

- Keep it simple—one column per attribute.
  - This means storing first name and last name as two separate fields if you need to query them independently.

```
CREATE TABLE users (
  . . .
  first_name VARCHAR(100) NOT NULL,
  last_name VARCHAR(100) NOT NULL
);
```

Don’t Reuse Columns

- Don’t reuse columns for different information.
  - Column `episode_or_season` is confusing, it means that the columns stores episode numbers *sometimes*.
  - You need to write more application code to parse data, or use an extra column to distinguish them.

```
CREATE TABLE title (
  . . .
  episode_or_season INT,
  which_is_it ENUM('episode', 'season')
);
```

One Value per "Cell"

- Store only one value in each column and row.
  - Don't use comma-separated lists or multiple columns.
  - For instance, if your user has more than one phone number, create another table.

```
CREATE TABLE phones (
  user_id INT NOT NULL,
  phone_number VARCHAR(20) NOT NULL,
  FOREIGN KEY (user_id)
   REFERENCES users(user_id)
);
```

Is a Comma-Separated List So Bad?

- Can’t ensure that each value is the right data type; no way to prevent *1,2,3,banana,5*
- Can’t use a foreign key constraint.
- Can’t use a unique constraint; no way to prevent *1,2,3,3,3,5*
- Can’t delete a value without fetching the whole list.
- Can’t query the count/average/max/min of elements in the list.
- Can’t fetch the list in sorted order.
- Hard to search for all rows with a given value in the list.
- Hard to join the values to the lookup table they reference.
- Storing integers as strings takes twice as much space.

Table Design Pitfalls

- Some physical realities affect table design choices:
  - Hot column on a wide table
  - Too many rows
  - Too many columns
  - Storage engine

Hot Column on a Wide Table

```
CREATE TABLE users (
  ID INTEGER,
  first_name VARCHAR(60),
  last_name VARCHAR(60),
  email VARCHAR(100),
  phone_number varchar(20),
  ...
  last_login_date DATE
);
```

this column is updated on every login

Hot Column on a Wide Table

- The problem:
  - Wide tables take more pages in memory. If we only need to update one column, it has a lot of overhead.
- The solution:
  - Create another table with just `last_login_date` and `user_id` referencing the `users` table.
  - Many narrow rows can fit on a single page.
  - Use a covering index (better for read-heavy situations).

Too Many Rows

- The problem:
  - Tables grow without bound over time—to millions or billions of rows.
  - Performance problems typically center around one or two very large tables.
  - Examples: “log” table, “events” table, “properties” table.

Too Many Rows

- Why this is a problem:
  - Queries and inserts get much slower once the table's indexes no longer fit in the buffer pool.
  - ALTER TABLE to add a column or index becomes very painful.

Too Many Rows

- The solutions:
  - Increase buffer pool size (can't do this forever)
  - Split the table!
- There are many strategies to split a table:
  - Use MySQL Partitioning
  - Implement your own partitioning in application code
  - Archive older data (use pt-archiver)

Too Many Columns

- The problem:
  - MySQL stores the columns of each row together. A query that only uses a few columns still needs to read the whole row.
  - This consumes a lot more I/O and buffer pool space, since the columns you need are stored sparsely among columns you don't need.

Too Many Columns

- The solutions:
  - Store some of the data in smaller tables (a table with so many columns might be improperly normalized).
  - If you find a lot of the additional columns are "optional attributes," then concatenate them together into one BLOB column (more on this later).

Too Many Columns

- The net wins from these solutions:
  - Fit more narrow rows per page.
  - Better utilization of buffer pool.
  - Reduced I/O when examining rows on risk.
  - If you're in the habit of using `SELECT * FROM table`, the query extracts less information (especially BLOBs).

Storage Engine

- InnoDB is recommended in most cases.
  - Active development and enhancements in every release.
  - Buffers both data and index pages.
  - Supports transactions.
  - Supports ACID, won't lose data in a crash.
- MyISAM is a legacy table type.
  - MySQL no longer devotes development resources to it.
  - Filesystem buffering only.
  - Non-atomic changes lead to corruption.

Schema Design

NORMALIZATION & DENORMALIZATION

What Is Normalization?

- A formal process for designing tables:
  - Eliminate redundant storage of data.
  - Prevent data anomalies.

Why Normalize?

- Best practice for organizing data, when you don't know which queries will be run against it.
- Data can't develop anomalies (orphaned rows, inconsistencies).
- You must understand the data relationships (even if you denormalize later).

Myths About Normalization

- Myth: "Normalization makes a database slower. Denormalization makes a database faster."
- Myth: "Normalization splits up tables as much as possible."
- Myth: "Normalization is using auto-increment primary keys."
- Myth: "Third normal form is enough for anyone."

First Normal Form

- Wrong:
  - Comma-separated lists are impossible to index.
  - Multi-column attributes (never enough columns).

movie_id	keyword_list	star1	star2	star3
207468	espionage, nuclear-bomb, ejector-seat	Sean Connery	Gert Fröbe	Honor Blackman

title

First Normal Form

- Right:

movie_id	title
207468	Goldfinger

title

movie_id	star
207468	Sean Connery
207468	Gert Fröbe
207468	Honor Blackman

cast_info

movie_id	keyword_list
207468	espionage
207468	nuclear-bomb
207468	ejector-seat

movie_keyword

Second Normal Form

- Wrong:
  - `movie_keyword` table with keyword as well as keyword_id can create inconsistencies:

movie_id	keyword_id	keyword
1234	6	silent-film
2345	6	silent-film
3456	6	silent-movie

movie_keyword

Second Normal Form

- Right:
  - Move keyword to its own table, where the keyword appears only once:

movie_id	keyword_id
1234	6
2345	6
3456	6

movie_keyword

keyword_id	keyword
5	hand-processed
6	silent-film
7	experimental-short

keyword

Third Normal Form

- Wrong:
  - Adding a column not related to the table's primary key.

movie_id	star	country
207468	Sean Connery	Scotland
207468	Gert Fröbe	Germany
226354	Sean Connery	Spain

cast_info

Third Normal Form

- Right:
  - Star attributes belong in the separate `name` table, so they appear only once, and can't have inconsistencies.

movie_id	star_id
207468	201693
207468	343779
226354	201693

cast_info

star_id	star	country
201693	Sean Connery	Scotland
343779	Gert Fröbe	Germany

name

Fourth Normal Form

- Wrong:
  - Store more than one many-to-many relationship in the same intersection table.
  - Leads to duplication of some values, or else use NULL.

movie_id	star	company	keyword
201693	Sean Connery	Eon Productions	espionage
201693	Gert Fröbe	Eon Productions	nuclear-bomb
201693	Honor Blackman	Eon Productions	ejector-seat

cast_info

Fourth Normal Form

- Right:
  - Store each many-to-many relationship in a separate intersection table.

movie_id	star
201693	Sean Connery
201693	Gert Fröbe
201693	Honor Blackman

cast_info

movie_id	company
201693	Eon Productions

movie_company

movie_id	keyword
201693	espionage
201693	nuclear-bomb
201693	ejector-seat

movie_keyword

What Is Denormalization?

- Judicious breaking of rules of normalization to get a performance gain for certain queries.

Why Denormalize?

- Basically, to reduce the work necessary during a query by pre-computing it.
  - Avoid expensive expressions.
  - Avoid expensive SUM, COUNT, AVG, etc.
  - Avoid expensive JOINs.
- This means you need to know what queries your users will execute.

Are There "*Denormal* Forms?"

- No formal definitions, but there are common patterns of denormalization:
  - Precalculated expressions
  - Materialized aggregates
  - Redundant columns

Precalculated Expressions

- Store results from expressions referencing other columns, to make it easier or faster to query.
  - E.g. extract a month from a date:

movie_id	title	release_date	release_month
207468	Goldfinger	1964-09-17	9
574127	Thunderball	1965-12-09	12

title

Materialized Aggregates

- Also called "summary table."
- Precalculate a `SUM()`, `COUNT()`, `AVG()`, `GROUP_CONCAT()` from related data.
  - E.g. `AVG(movie_ratings.rating)`

movie_id	title	release_date	rating_avg
207468	Goldfinger	1964-09-17	7.8
574127	Thunderball	1965-12-09	7.0

title

Redundant Columns

- Avoid a `JOIN` by copying frequently-used columns from related table(s).

id	person_id	name	movie_id	title	person_role_id	role_name
1514397	201693	Connery, Sean	207468	Goldfinger	35721	James Bond
2601856	343779	Gert Fröbe	207468	Goldfinger	366927	Auric Goldfinger

cast_info

Schema Design

INDEX DESIGN

Over-Indexed Tables

- Infrequently used indexes can be responsible for decreasing write capacity.
- It also increases memory and storage requirements.
- For reads, the optimizer has more choices to make and a more difficult decision process.

Under-Indexed Tables

- Under-indexed tables can result in too many rows needing to be examined after an index has been used—or in the worst case, no index used.
  - This can cause contention on what contents you are able to keep in memory—and it will likely increase the size of your working set.

How to Find Duplicate Indexes

- pt-duplicate-key-checker

```
# ########################################################################
# wordpress.wp_posts
# ########################################################################

# Key type_status_date ends with a prefix of the clustered index
# Key definitions:
#   KEY `type_status_date` (`post_type`,`post_status`,`post_date`,`ID`),
#   PRIMARY KEY (`ID`),
# Column types:
#         `post_type` varchar(20) not null default 'post'
#         `post_status` varchar(20) not null default 'publish'
#         `post_date` datetime not null default '0000-00-00 00:00:00'
#         `id` bigint(20) unsigned not null auto_increment
# To shorten this duplicate clustered index, execute:
ALTER TABLE `wordpress`.`wp_posts` DROP INDEX `type_status_date`, ADD INDEX
`type_status_date` (`post_type`,`post_status`,`post_date`);
```

How to Find Unused Indexes

- pt-index-usage

```
$ pt-index-usage slow.log
slow.log: 11% 03:58 remain
slow.log: 21% 03:43 remain
slow.log: 32% 03:09 remain
[...]
ALTER TABLE `tpcc`.`order_line` DROP KEY `fkey_order_line_2`; --
type:non-unique
ALTER TABLE `tpcc`.`orders` DROP KEY `idx_orders`; -- type:non-unique
ALTER TABLE `tpcc`.`stock` DROP KEY `fkey_stock_2`; -- type:non-unique
```

How to Find Unused Indexes

- userstats in Percona Server (Google Patches)
  - `INFORMATION_SCHEMA.INDEX_STATISTICS`
- PERFORMANCE_SCHEMA in MySQL 5.6
  - `table_io_waits_summary_by_index_usage`

http://www.mysqlperformanceblog.com/2008/09/12/unused-indexes-by-single-query/

Designing for Memory Fit

- Index performance degrades sharply if indexes are too large to fit in memory.
- Initially the size of your indexes is small, and they fit in memory easily.
  - Is that assumption going to remain true?
  - If YES—keep in mind the maximum size of indexes.
  - If NO—you need to test with synthetic data to make sure performance does not suffer as data grows.

Schema Design

SQL CONSTRAINTS

Primary Keys

- The purpose of a primary key is to allow you to uniquely reference individual rows.

```
CREATE TABLE movie_keyword (
  id int NOT NULL AUTO_INCREMENT,
  movie_id int NOT NULL,
  keyword_id int NOT NULL,
  PRIMARY KEY (id)
);
DELETE FROM movie_keyword
WHERE id = 883543;
```

Auto-Increment Primary Keys

- Using auto-increment generates a new value at the end of the clustered index.

```
CREATE TABLE movie_keyword (
  id int NOT NULL AUTO_INCREMENT,
  movie_id int NOT NULL,
  keyword_id int NOT NULL,
  PRIMARY KEY (id)
);
INSERT INTO movie_keyword
(movie_id,keyword_id) VALUES (207468,5467);
```

Compound Primary Keys

- Compound primary keys are okay too.
  - You can still reference rows uniquely.

```
CREATE TABLE movie_keyword (
  movie_id int NOT NULL,
  keyword_id int NOT NULL,
  PRIMARY KEY (movie_id, keyword_id)
);
DELETE FROM movie_keyword
WHERE (movie_id,keyword_id) = (207468,5467);
```

Optimizing Primary Keys

- Match to your most common query if possible.

```
CREATE TABLE movie_keyword (
  movie_id int NOT NULL,
  keyword_id int NOT NULL,
  PRIMARY KEY (keyword_id, movie_id)
);
```

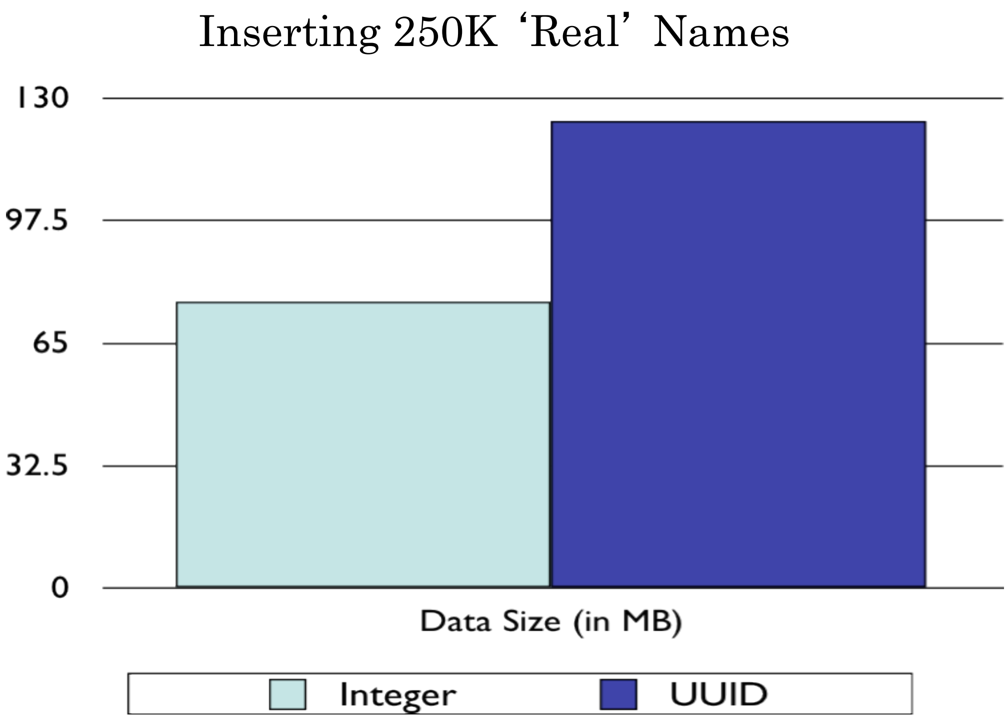
```
SELECT * FROM movie_keyword
WHERE keyword_id = 5467;
```

Piggy-Back on Secondary Keys

- Every InnoDB secondary key implicitly contains the primary key.

```
CREATE TABLE title (
  id int NOT NULL AUTO_INCREMENT,
  title text NOT NULL,
  PRIMARY KEY (id),
  KEY (title(50)) /* also includes id */
) ENGINE=InnoDB;
```

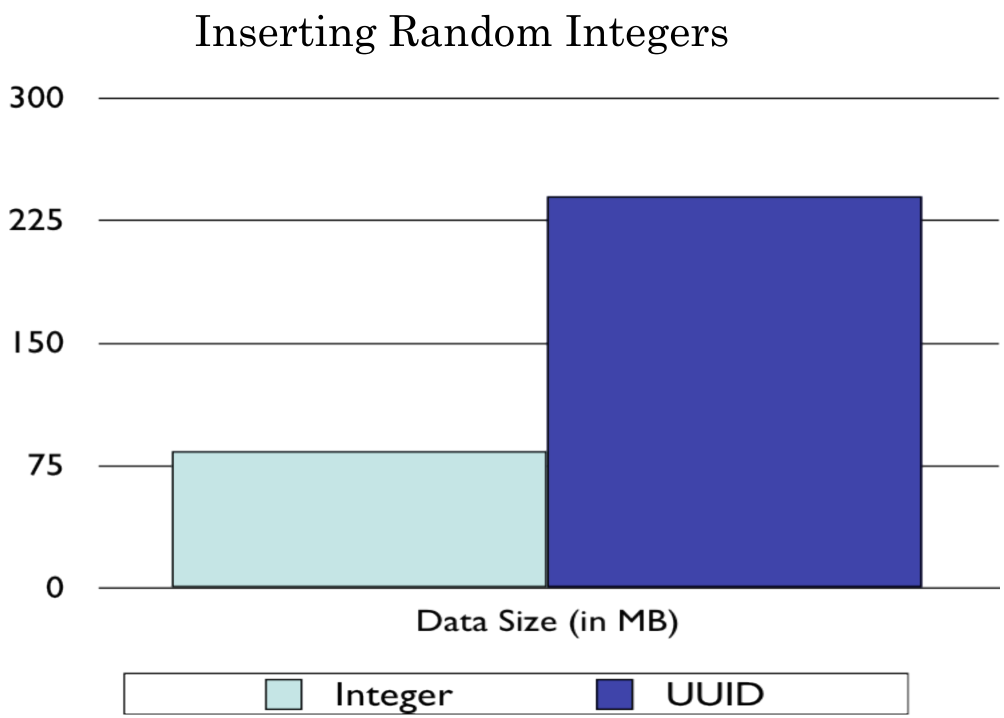
Our Results (Typical Case)




```
CREATE TABLE uuid_users (
  PRIMARY KEY,
  emailaddress varchar(100),
  firstname varchar(20),
  lastname varchar(20),
  birthday varchar(10),
  occupation varchar(70), INDEX
  (emailaddress),
  INDEX(lastname, firstname),
  INDEX(occupation)
) ENGINE=InnoDB;
```

The UUID primary key makes the table about 65% larger.

Our Results (Worst Case)




```
CREATE TABLE mydata (
  PRIMARY KEY,
  col1 INT NOT NULL,
  col2 INT NOT NULL,
  col3 INT NOT NULL,
  col4 INT NOT NULL,
  col5 INT NOT NULL,
  INDEX (col1),
  INDEX (col2),
  INDEX (col3),
  INDEX (col4),
  INDEX (col5)
) ENGINE=InnoDB;
```

The UUID primary key makes the table almost x3!

Primary Key Data Types

- Choose the most compact data type that fits.
  - `SMALLINT` or `INT` or `BIGINT` for an auto-increment.
- Avoid long strings if possible.
  - For example, UUID's are often used for uniqueness across clusters, but they are long strings. Also, UUID's insert in random order, causing fragmentation.

Unique Keys

- Used if other column or columns need to be unique.
  - You can have multiple unique keys per table.
  - May be NULL, whereas primary keys cannot.

```
CREATE TABLE movie_keyword (
  id int NOT NULL AUTO_INCREMENT,
  movie_id int NOT NULL,
  keyword_id int NOT NULL,
  PRIMARY KEY (id),
  UNIQUE KEY (keyword_id, movie_id)
);
```

Foreign Keys

- Important to enforce referential integrity.

```
CREATE TABLE movie_keyword (
  movie_id int NOT NULL,
  keyword_id int NOT NULL,
  PRIMARY KEY (keyword_id, movie_id),
  FOREIGN KEY (movie_id)
    REFERENCES title(id),
  FOREIGN KEY (keyword_id)
    REFERENCES keyword(id)
);
```

Fun Foreign Key Facts

- Only InnoDB supports foreign keys.
  - MyISAM parses but *ignores* foreign keys.
- Foreign keys can reference tables across databases.
- Foreign keys can reference either a primary key or a unique key.
- The data type of a foreign key must match its referenced column *exactly*—type, sign, precision, character set.
  - Except for varchar length.

Check Constraints

- Enforces some business rules, e.g. postal codes must match a pattern of numbers and/or letters.
- MySQL parses but *ignores* CHECK constraints.
- Workarounds:
  - Foreign keys to a lookup table
  - `ENUM()` data type
  - Triggers

Nullability

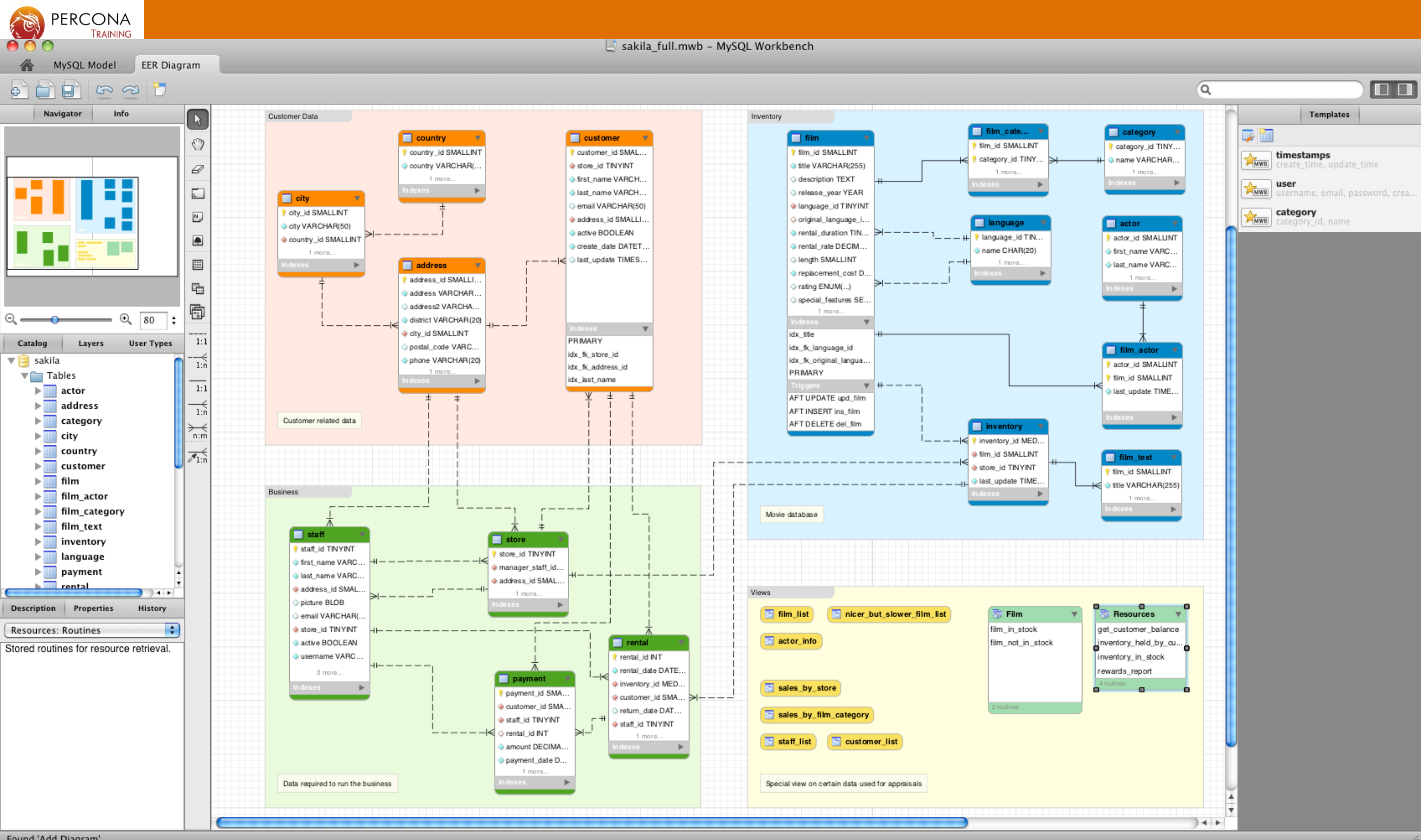
- `NULL` is for missing or unknown values.
  - If the column is mandatory, declare it `NOT NULL`.
  - No significant performance cost; declare `NOT NULL` for reasons of logic, not optimization.
- Don't use `-1` or `'N/A'` or other "special" value to signify a missing value.
  - This confuses expressions, aggregates, etc.

Schema Design

SCHEMA DESIGN TOOLS

Schema Design Tools

- Use a program where you can map out each of the objects on an Entity-Relationship (ER) diagram.
  – MySQL Workbench.
- Export the ER diagram to SQL.


PERCONA
TRAINING
sakila_full.mwb – MySQL Workbench
MySQL Model
EER Diagram
Customer Data
Inventory
Business
Views
Customer related data
Movie database
Data required to run the business
Special view on certain data used for appraisals
country
customer
city
address
film
film_category
category
language
actor
film_actor
inventory
film_text
staff
store
rental
payment
film_list
nicer_but_slower_film_list
actor_info
sales_by_store
sales_by_film_category
staff_list
customer_list
Film
film_in_stock
film_not_in_stock
Resources
get_customer_balance
inventory_in_stock
rewards_report
Templates
timestamps
user
category
sakila
Tables
Resources: Routines
Stored routines for resource retrieval.
Found 'Add Diagram'

Can You Make a Schema Better?

- It's very hard to retrofit into an application.
- For some obvious bad-choices, the "band aid" approach may work.
  - This command shows the most optimal data type:
  - `SELECT * FROM title PROCEDURE ANALYSE(1,1)`

http://dev.mysql.com/doc/refman/5.6/en/procedure-analyse.html

Schema Design

VIEWS

Views

- Create some ‘hidden’ work. Some views need to create temporary tables.
- No indexes on views.
- No materialized views

Schema Design

STORED ROUTINES

Stored Routines in MySQL

- You can write blocks of code in an SQL-like language that runs in the database server.
- Your application can invoke stored procedures with `CALL` *`ProcedureName`*`()`.
- Your application can invoke stored functions in any SQL expression like `SELECT` *`FunctionName`*`()...`

Stored Routine Culture

- Using stored routines is traditional in commercial enterprise databases, but MySQL's implementation is primitive by comparison.
- They have their place, but don't plan to use them extensively. *MySQL is not Oracle.*

Cons of MySQL Stored Routines (1)

- They are not compiled. They are parsed on first execution in each connection.
- There are no debugger features.
- Language is limited and not extensible.
- No packages, libraries, or inheritance features.

Cons of MySQL Stored Routines (2)

- Procedures with dynamic SQL are unsafe for statement based replication.
- Query logs contain the CALL but not the individual statements run inside the routine.
  - Percona Server has an option to fix this: `log_slow_sp_statements` http://www.percona.com/doc/percona-server/5.5/diagnostics/slow_extended_55.html

Pros of MySQL Stored Routines

- You can write procedures that performs a complex, multi-step task, and avoid transferring query results across the network.
- You can grant limited privileges to a MySQL user, but allow them privilege to call a procedure that has more privileges itself.

Events

- No (built-in) protection against events running simultaneously.
- Same restrictions as routines – very hard to get consistently working with Replication.

Schema Design

TRIGGERS

Triggers

- Similar limitations to Stored Procedures.
- Only one for-each-row trigger per event.
- Triggers create a lot of 'hidden work', and increase server locking.
- No elegant way to disable triggers for data reload / batch operations.
- Behavior is different between statement and row based replication.

Schema Design

OBJECT-RELATIONAL MAPPING

ORM Pros

- Let developers access data in a familiar object-oriented paradigm, instead of writing SQL.
- Improve developer productivity. E.g., eliminate repetitive code to copy query result fields to object members and vice-versa.
- Infer object relationships from SQL metadata.
- Generate boilerplate code for basic queries.

ORM Cons

- Fails to handle all SQL query constructs (JOIN types, GROUP BY, HAVING, DISTINCT, etc.)
- Some ORM frameworks get metadata from XML instead of inspecting the database. If you alter the schema, you must also update the ORM.
- Queries generated by an ORM are sometimes ridiculously inefficient (e.g. dozens of joins).
- Abstract table definition makes bad choices about datatypes (e.g. VARCHAR(255) for everything).

ORM Cons (cont.)

- ORM hides rampantly inefficient patterns of querying.

```
/* query for one order and its children */
$order = Order::find(1234);
$items = $order->getLineItems();

/* query for N multiple orders, and run N
queries for children of each order */
$orders = Order::findByUser('bill');
$items = $orders->getAllLineItems();
```

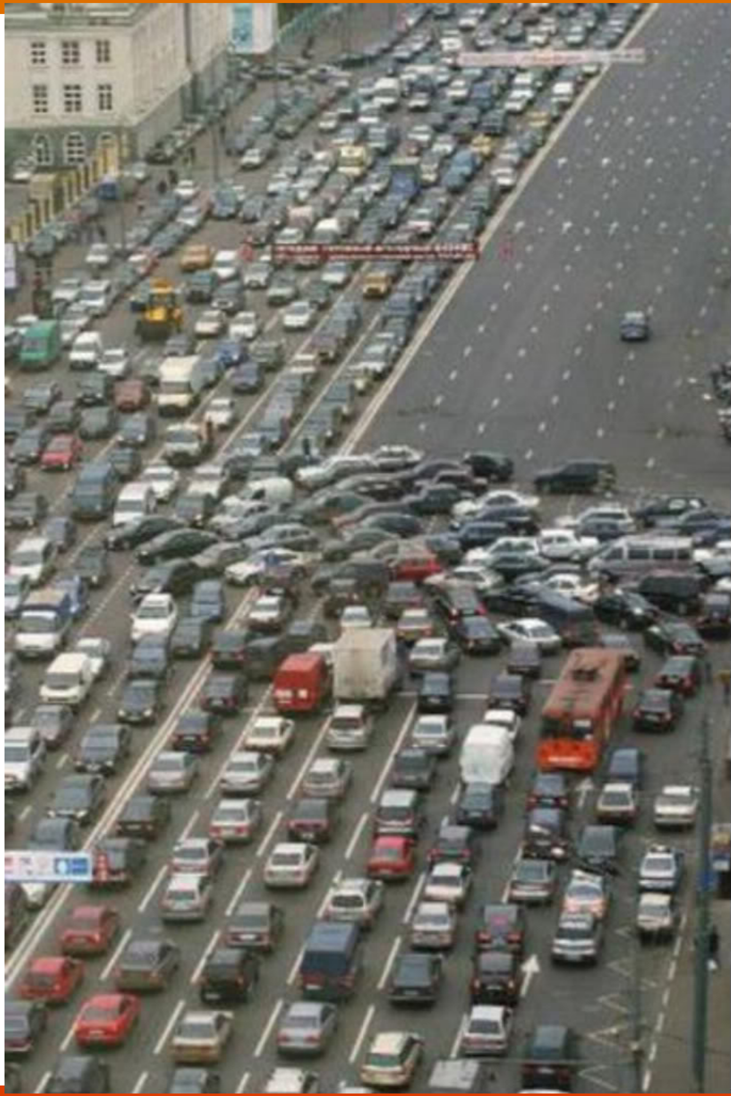
What to Do?

- Use the ORM initially, to gain the developer productivity advantage.
- Use available options of the ORM, so you aren't relying on default (inefficient) behavior.
- Later, as you measure performance bottlenecks, unravel your ORM usage, replacing it with hand-coded SQL.
  - Just like a systems programmer would optimize C code with blocks of assembly language.

Schema Design

EXTENSIBLE SCHEMA DESIGN

“I need to add a new column—but I don’t want ALTER TABLE to lock the application for a long time.”



How MySQL Does ALTER TABLE

1. Lock the table.
2. Make a new, empty the table like the original.
3. Modify the columns of the new empty table.
4. Copy all rows of data from original to new table…
   no matter how long it takes.
5. Swap the old and new tables.
6. Unlock the tables & drop the original.

Extensibility

- How can we add new attributes without the pain of schema changes?
  - Object-oriented modeling
  - Sparse columns
- Especially to support *user-defined* attributes at runtime or after deployment:
  - Content management systems
  - E-commerce frameworks
  - Games

Solutions

- “Extra Columns”
- Entity-Attribute-Value
- Class Table Inheritance
- Serialized LOB & Inverted Indexes
- Online Schema Changes
- Non-Relational Databases

Extensible Schema Design

EXTRA COLUMNS

Table with Fixed Columns

```
CREATE TABLE Title (
  id int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
  title text NOT NULL,
  imdb_index varchar(12) DEFAULT NULL,
  kind_id int(11) NOT NULL,
  production_year int(11) DEFAULT NULL,
  imdb_id int(11) DEFAULT NULL,
  phonetic_code varchar(5) DEFAULT NULL,
  episode_of_id int(11) DEFAULT NULL,
  season_nr int(11) DEFAULT NULL,
  episode_nr int(11) DEFAULT NULL,
  series_years varchar(49) DEFAULT NULL,
  title_crc32 int(10) unsigned DEFAULT NULL
);
```

Table with Extra Columns

```
CREATE TABLE Title (
  id int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
  title text NOT NULL,
  imdb_index varchar(12) DEFAULT NULL,
  kind_id int(11) NOT NULL,
  production_year int(11) DEFAULT NULL,
  imdb_id int(11) DEFAULT NULL,
  phonetic_code varchar(5) DEFAULT NULL,
  extra_data1 TEXT DEFAULT NULL,
  extra_data2 TEXT DEFAULT NULL,
  extra_data3 TEXT DEFAULT NULL,
  extra_data4 TEXT DEFAULT NULL,
  extra_data5 TEXT DEFAULT NULL,
  extra_data6 TEXT DEFAULT NULL,
);
```

use for whatever comes up that we didn't think of at the start of the project

Adding a New Attribute

```
UPDATE Title
SET extra_data3 = 'PG-13'
WHERE id = 207468;
```

remember which column you used for each new attribute!

Pros and Cons

- Good solution:
  - No ALTER TABLE necessary to use a column for a new attribute—only a project decision is needed.
  - Related to Single Table Inheritance (STI) http://martinfowler.com/eaaCatalog/singleTableInheritance.html

Pros and Cons

- Bad solution:
  - If you run out of extra columns, then you're back to ALTER TABLE.
  - Anyone can put any data in the columns—you can't assume consistent usage on every row.
  - Columns lack descriptive names or the right data type.

Extensible Schema Design

ENTITY-ATTRIBUTE-VALUE

EAV

- Store each attribute in a row instead of a column.

```
CREATE TABLE Attributes (
  entity     INT NOT NULL,
  attribute VARCHAR(20) NOT NULL,
  value      TEXT,
  FOREIGN KEY (entity)
   REFERENCES Title (id)
);
```

Example EAV Data

```
SELECT * FROM Attributes;
+--------+-----------------+---------------------+
| entity | attribute       | value               |
+--------+-----------------+---------------------+
| 207468 | title           | Goldfinger          |
| 207468 | production_year | 1964                |
| 207468 | rating          | 7.8                 |
| 207468 | length          | 110 min             |
+--------+-----------------+---------------------+
```

Adding a New Attribute

- Simply use INSERT with a new attribute name.

```
INSERT INTO Attributes (entity, attribute, value)
VALUES (207468, 'budget', '$3,000,000');
```

Query EAV as a Pivot

```
SELECT a.entity AS id,
       a.value AS title,
       y.value AS production_year,
       r.value AS rating,
       b.value AS budget
FROM Attributes AS a
JOIN Attributes AS y USING (entity)
JOIN Attributes AS r USING (entity)
JOIN Attributes AS b USING (entity)
WHERE a.attribute = 'title'
  AND y.attribute = 'production_year'
  AND r.attribute = 'rating'
  AND b.attribute = 'budget';
+--------+------------+-----------------+--------+------------+
| id     | title      | production_year | rating | budget     |
+--------+------------+-----------------+--------+------------+
| 207468 | Goldfinger | 1964            | 7.8    | $3,000,000 |
+--------+------------+-----------------+--------+------------+
```

another join required for each additional attribute

Sounds Simple Enough, But…

- NOT NULL doesn't work
- FOREIGN KEY doesn't work
- UNIQUE KEY doesn't work
- Data types don't work
- Searches don't scale
- Indexes and storage are inefficient

Constraints Don't Work

```
CREATE TABLE Attributes (
  entity     INT NOT NULL,
  attribute VARCHAR(20) NOT NULL,
  value      TEXT NOT NULL,
  FOREIGN KEY (entity)
   REFERENCES Title (id)
  FOREIGN KEY (value)
   REFERENCES Ratings (rating)
);
```

constraints apply to all rows, not just rows for a specific attribute type

Data Types Don't Work

```
INSERT INTO Attributes (entity, attribute, value)
VALUES (207468, 'budget', 'banana');
```

the database can't prevent the application from storing nonsense data

Add Typed Value Columns?

```
CREATE TABLE Attributes (
  entity        INT NOT NULL,
  attribute     VARCHAR(20) NOT NULL,
  intvalue      BIGINT,
  floatvalue    FLOAT,
  textvalue     TEXT,
  datevalue     DATE,
  datetimevalue DATETIME,
  FOREIGN KEY (entity)
   REFERENCES Title (id)
);
```

now the application needs to know which data type column to use for each attribute when inserting and querying

Searches Don't Scale

- You must hard-code each attribute name,
  - One `JOIN` per attribute!

- Alternatively, you can query all attributes, but the result is one attribute per row:

```
SELECT attribute, value
FROM Attributes
WHERE entity = 207468;
```

  - ...and sort it out in your application code.

Indexes and Storage Are Inefficient

- Many rows, with few distinct `attribute` names.
  - Poor index cardinality.
- The `entity` and `attribute` columns use extra space for every attribute of every “row.”
  - In a conventional table, the *entity* is the primary key, so it’s stored only once per row.
  - The attribute *name* is in the table definition, so it’s stored only once per table.

Pros and Cons

- Good solution:
  - No ALTER TABLE needed again—ever!
  - Supports ultimate flexibility, potentially any "row" can have its own distinct set of attributes.

Pros and Cons

- Bad solution:
  - SQL operations become more complex.
  - Lots of application code required to reinvent features that an RDBMS already provides.
  - Doesn't scale well—pivots required.

Extensible Schema Design

CLASS TABLE INHERITANCE

Subtypes

- Titles includes:
  - Films
  - TV shows
  - TV episodes
  - Video games
- Some attributes apply to all, other attributes apply to one subtype or the other.

Title Table

```
CREATE TABLE Title (
  id int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
  title text NOT NULL,
  imdb_index varchar(12) DEFAULT NULL,
  kind_id int(11) NOT NULL,
  production_year int(11) DEFAULT NULL,
  imdb_id int(11) DEFAULT NULL,
  phonetic_code varchar(5) DEFAULT NULL,
  episode_of_id int(11) DEFAULT NULL,
  season_nr int(11) DEFAULT NULL,
  episode_nr int(11) DEFAULT NULL,
  series_years varchar(49) DEFAULT NULL,
  title_crc32 int(10) unsigned DEFAULT NULL
);
```

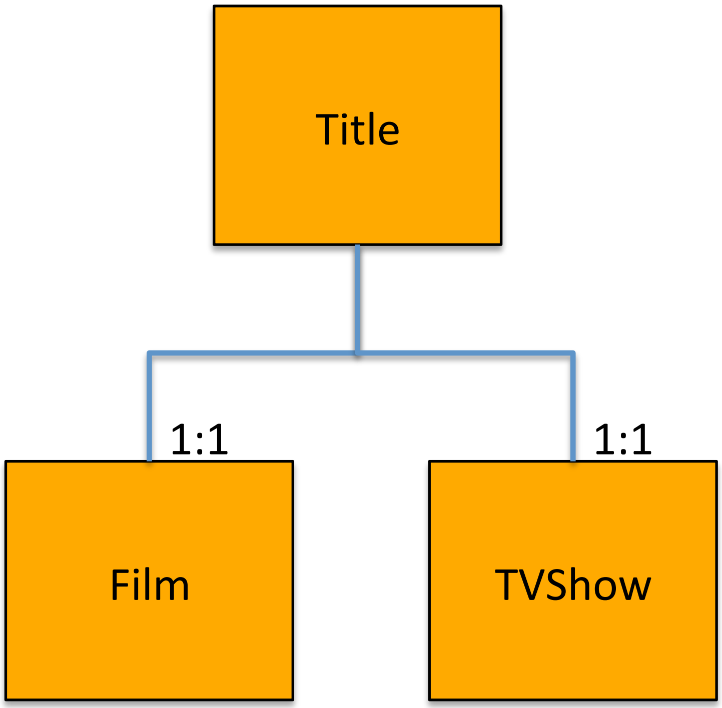
only for tv shows

Title Table with Subtype Tables

```
CREATE TABLE Title (
  id int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
  title text NOT NULL,
  imdb_index varchar(12) DEFAULT NULL,
  kind_id int(11) NOT NULL,
  production_year int(11) DEFAULT NULL,
  imdb_id int(11) DEFAULT NULL,
  phonetic_code varchar(5) DEFAULT NULL,
  title_crc32 int(10) unsigned DEFAULT NULL,
  PRIMARY KEY (id)
);

CREATE TABLE Film (
  id int(11) NOT NULL PRIMARY KEY,
  aspect_ratio varchar(20),
  FOREIGN KEY (id) REFERENCES Title(id)
);

CREATE TABLE TVShow (
  id int(11) NOT NULL PRIMARY KEY,
  episode_of_id int(11) DEFAULT NULL,
  season_nr int(11) DEFAULT NULL,
  episode_nr int(11) DEFAULT NULL,
  series_years varchar(49) DEFAULT NULL,
  FOREIGN KEY (id) REFERENCES Title(id)
);
```



Adding a New Subtype

- Create a new table—without locking existing tables.

```
CREATE TABLE VideoGames (
  id int(11) NOT NULL PRIMARY KEY,
  platforms varchar(100) NOT NULL,
  FOREIGN KEY (id)
   REFERENCES Title(id)
);
```

Pros and Cons

- Good solution:
  - Best to support a finite set of subtypes, which are likely unchanging after creation.
  - Data types and constraints work normally.
  - Easy to create or drop subtype tables.
  - Easy to query attributes common to all subtypes.
  - Subtype tables are shorter, indexes are smaller.

Pros and Cons

- Bad solution:
  - Adding one entry takes two INSERT statements.
  - Querying attributes of subtypes requires a join.
  - Querying all types with subtype attributes requires multiple joins (as many as subtypes).
  - Adding a common attribute locks a large table.
  - Adding an attribute to a populated subtype locks a smaller table.

Extensible Schema Design

SERIALIZED LOB

What is Serializing?

- Objects in your applications can be represented in *serialized* form—i.e., convert the object to a scalar string that you can save and load back as an object.
  – Java objects implementing `Serializable` and processed with `writeObject()`
  – PHP variables processed with `serialize()`
  – Python objects processed with `pickle.dump()`
  – Data encoded with XML, JSON, YAML, etc.

What Is a LOB?

- The BLOB or TEXT datatypes can store long sequences of bytes or characters, such as a string.
- You can store the string representing your object into a single BLOB or TEXT column.
  - You don't need to define SQL columns for each field of your object.

Title Table with Serialized LOB

```
CREATE TABLE Title (
  id int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
  title text NOT NULL,
  imdb_index varchar(12) DEFAULT NULL,
  kind_id int(11) NOT NULL,
  production_year int(11) DEFAULT NULL,
  imdb_id int(11) DEFAULT NULL,
  phonetic_code varchar(5) DEFAULT NULL,
  title_crc32 int(10) unsigned DEFAULT NULL
  extra_info TEXT
);
```

holds everything else, plus anything we didn't think of

Adding a New Attribute

```
UPDATE Title
SET extra_info =
  '{
    "episode_of_id": "1291895",
    "season_nr": "5",
    "episode_nr": "6"
   }'
WHERE id = 1292057;
```

Using XML in MySQL

- MySQL has limited support for XML.

```
SELECT id, title,
  ExtractValue(extra_info, '/episode_nr')
    AS episode_nr
FROM Title
WHERE ExtractValue(extra_info,
  '/episode_of_id') = 1292057;
```

- Forces table-scans, not possible to use indexes.

http://dev.mysql.com/doc/refman/5.6/en/xml-functions.html

Pros and Cons

- Good solution:
  - Store any object and add new custom fields at any time.
  - No need to do ALTER TABLE to add custom fields.

Pros and Cons

- Bad solution:
  - Not indexable.
  - Must return the whole object, not an individual field.
  - Must write the whole object to update a single field.
  - Hard to use a custom field in a `WHERE` clause, `GROUP BY` or `ORDER BY`.
  - No support in the database for data types or constraints, e.g. `NOT NULL`, `UNIQUE`, `FOREIGN KEY`.

Extensible Schema Design

ONLINE SCHEMA CHANGES

pt-online-schema-change

- Performs online, non-blocking ALTER TABLE.
  - Captures concurrent updates to a table while restructuring.
  - Some risks and caveats exist; please read the manual and test carefully.
- Free tool—part of Percona Toolkit.
  - http://www.percona.com/doc/percona-toolkit/pt-online-schema-change.html

How MySQL Does ALTER TABLE

1. Lock the table.
2. Make a new, empty the table like the original.
3. Modify the columns of the new empty table.
4. Copy all rows of data from original to new table.
5. Swap the old and new tables.
6. Unlock the tables & drop the original.

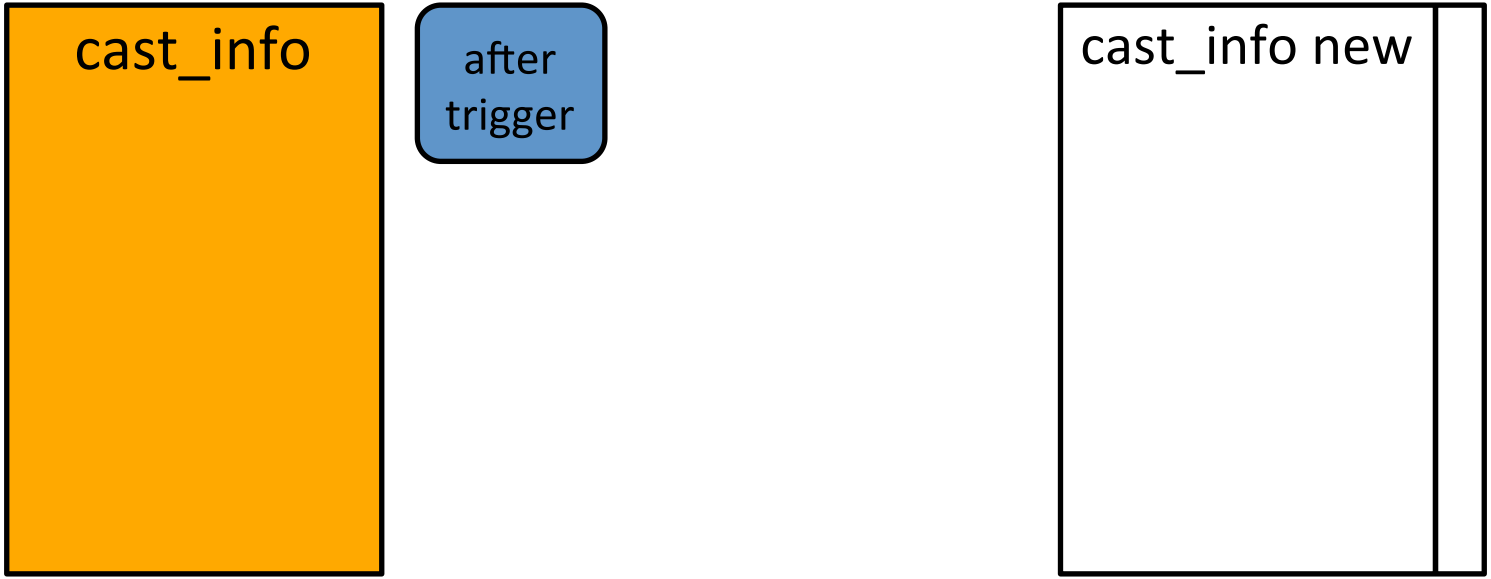
How pt-osc Does ALTER TABLE

~~Lock the table.~~

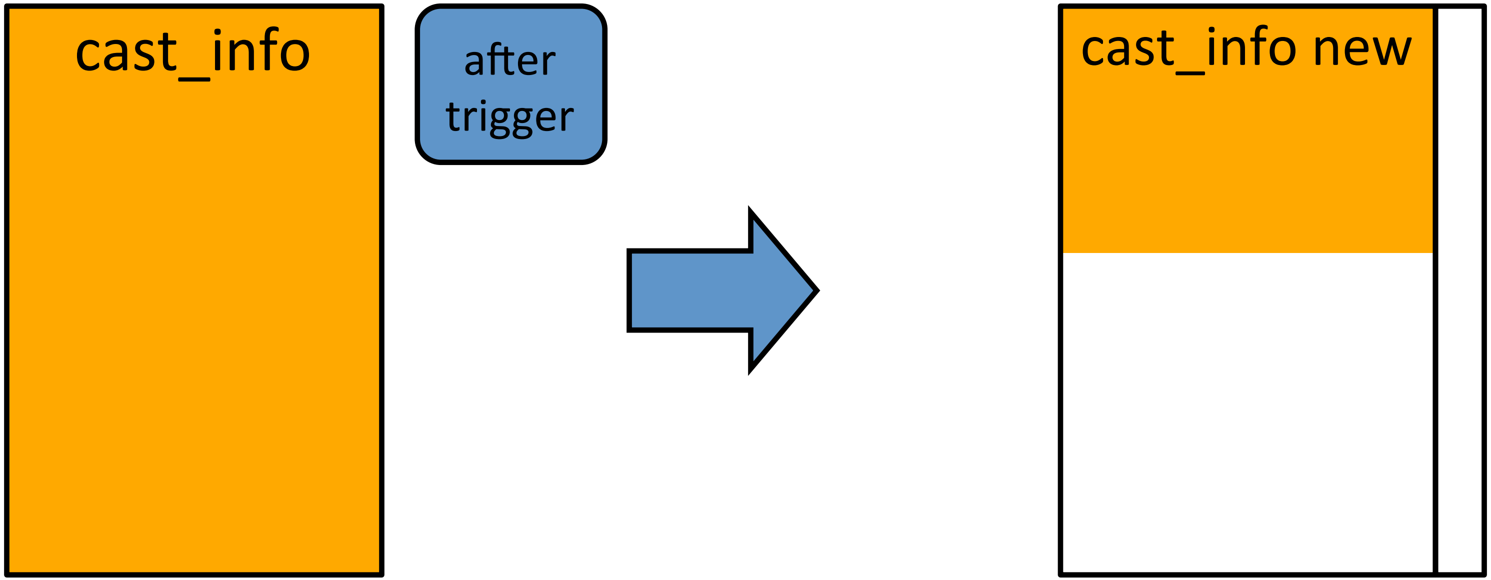
1. Make a new, empty the table like the original.
2. Modify the columns of the new empty table.
3. Copy all rows of data from original to new table.
    a. Iterate over the table in chunks, in primary key order.
    b. Use triggers to capture ongoing changes in the original, and apply them to the new table.
4. Swap the tables, then drop the original.

~~Unlock the tables.~~

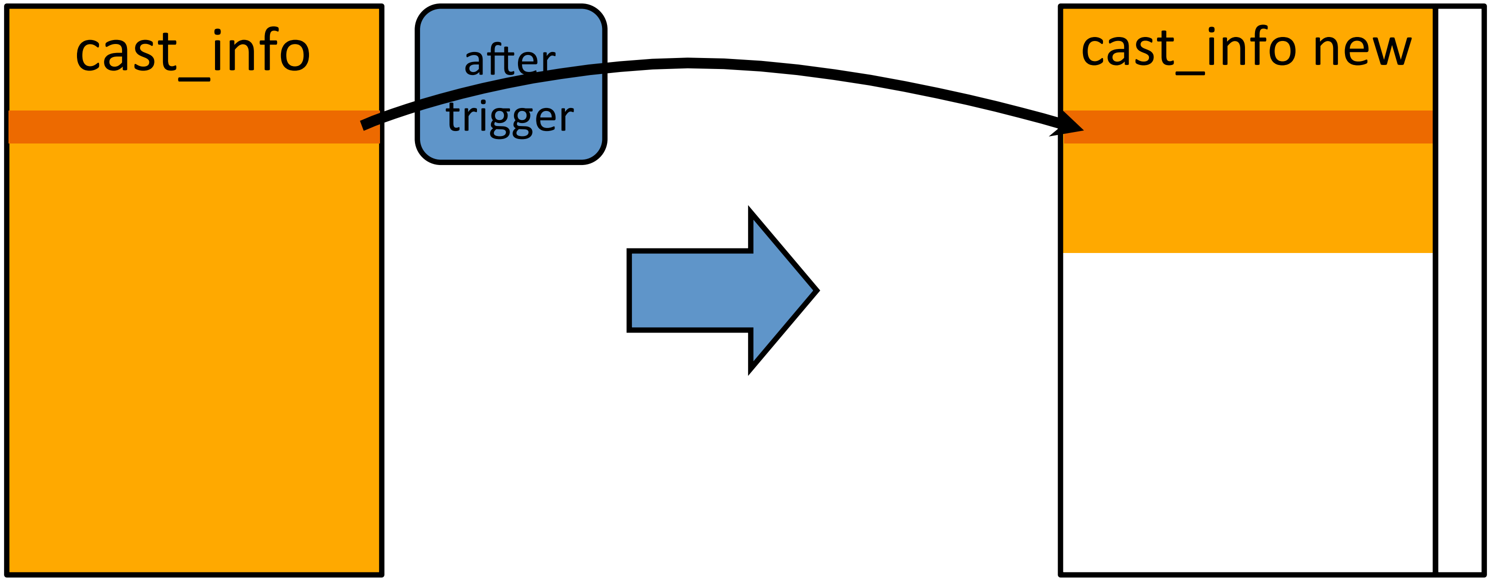
Visualize This (1)



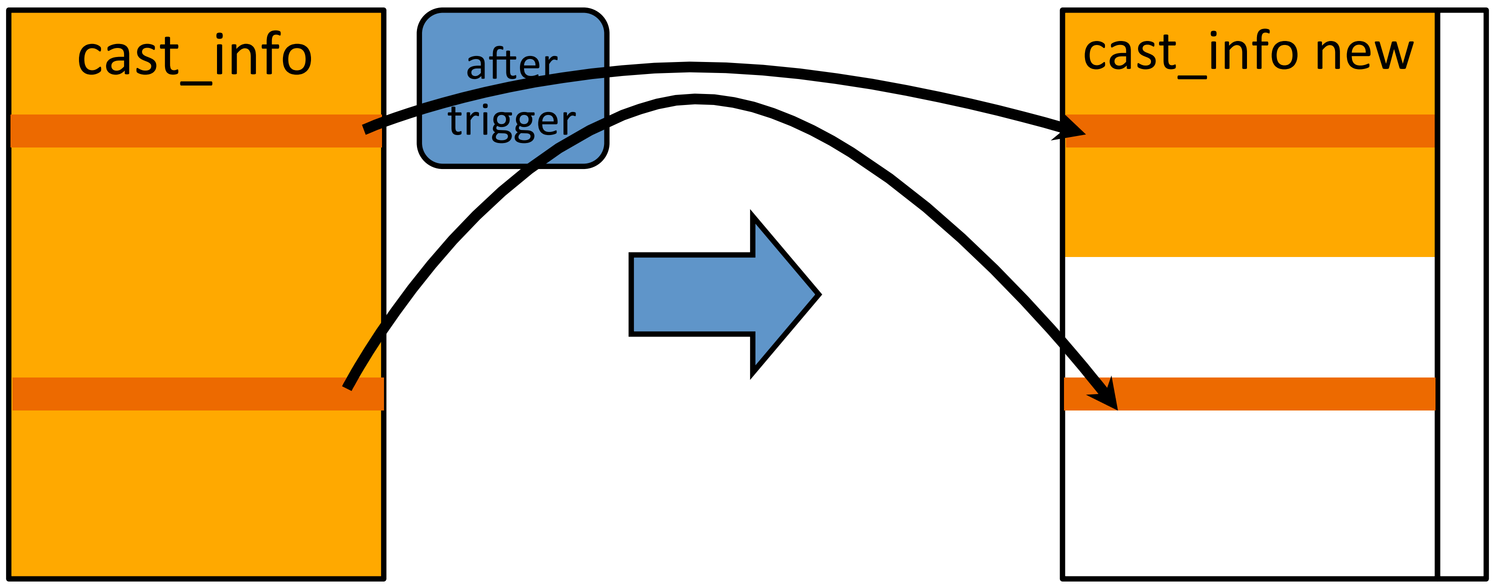
Visualize This (2)



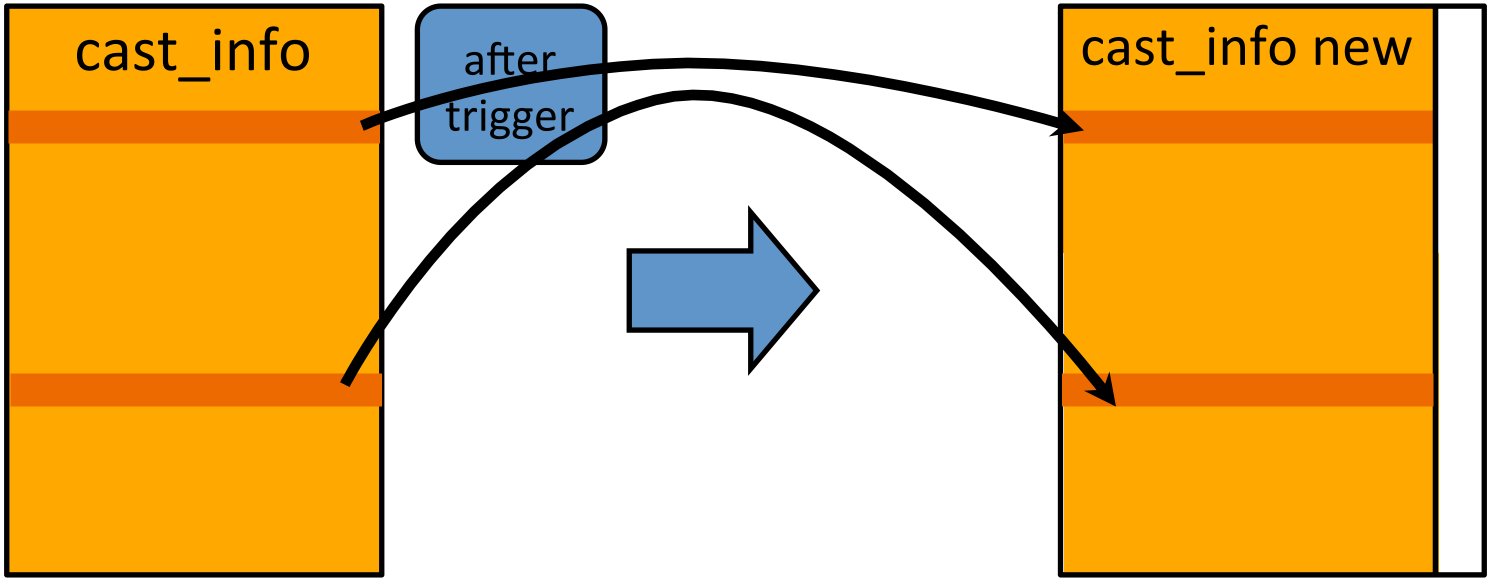
Visualize This (3)



Visualize This (4)



Visualize This (5)



Visualize This (6)

Adding a New Attribute

- Design the ALTER TABLE statement, but don't execute it yet.

```
mysql> ALTER TABLE cast_info
ADD COLUMN source INT NOT NULL;
```

- Equivalent pt-online-schema-change command:

```
$ pt-online-schema-change
  h=localhost,D=imdb,t=cast_info
  --alter "ADD COLUMN source INT NOT NULL"
```

Execute

```
$ pt-online-schema-change h=localhost,D=imdb,t=cast_info
  --alter "ADD COLUMN source INT NOT NULL" --execute
```

```
Altering `imdb`.`cast_info`...
Creating new table...
Created new table imdb._cast_info_new OK.
Altering new table...
Altered `imdb`.`_cast_info_new` OK.
Creating triggers...
Created triggers OK.
Copying approximately 22545051 rows...
Copying `imdb`.`cast_info`:  10% 04:05 remain
Copying `imdb`.`cast_info`:  19% 04:07 remain
Copying `imdb`.`cast_info`:  28% 03:44 remain
Copying `imdb`.`cast_info`:  37% 03:16 remain
Copying `imdb`.`cast_info`:  47% 02:47 remain
Copying `imdb`.`cast_info`:  56% 02:18 remain
Copying `imdb`.`cast_info`:  64% 01:53 remain
Copying `imdb`.`cast_info`:  73% 01:28 remain
Copying `imdb`.`cast_info`:  82% 00:55 remain
Copying `imdb`.`cast_info`:  91% 00:26 remain
Copied rows OK.
Swapping tables...
Swapped original and new tables OK.
Dropping old table...
Dropped old table `imdb`.`_cast_info_old` OK.
Dropping triggers...
Dropped triggers OK.
Successfully altered `imdb`.`cast_info`.
```


Self-Adjusting

- Copies rows in “chunks” which the tool sizes dynamically.
- The tool throttles back if it increases load too much or if it causes any replication slaves to lag.
- The tool tries to set its lock timeouts to let applications be more likely to succeed.

Why Shouldn't I Use This?

- Is your table small enough that `ALTER` is already quick enough?
- Is your change already very quick, for example `DROP KEY` in InnoDB?
- Will pt-online-schema-change take too long or increase the load too much?

Pros and Cons

- Good solution:
  - ALTER TABLE to add conventional columns without the pain of locking.

Pros and Cons

- Bad solution:
  - Can take up to 4× more time than ALTER TABLE.
  - Table must have a PRIMARY key.
  - Table must not have triggers.
  - No need if your table is small and ALTER TABLE already runs quickly enough.
  - No need for some ALTER TABLE operations that don't restructure the table (e.g. dropping indexes, adding comments).

Extensible Schema Design

NON-RELATIONAL ALTERNATIVES

Non-Relational Alternatives

- The basic rules of relational tables don't apply.
  - No table heading.
  - No data types.
  - No constraints.
  - Each "row" can have its own column names.
  - Cells may contain structured data: lists, collections, sub-cells, etc.

Pros and Cons

- Good solution:
  - You gain freedom from schema constraints.
- Bad solution:
  - You lose the benefits of schema constraints.
  - Your application can no longer assume all entries have the same structure.
  - You have to write a lot more code to inspect each entry.
  - Altering the structure is just as complex if you have to alter historical data retroactively.

Summary

Solution	Lock-free	Flexible	Select	Filter	Indexed	Data Types	Constraints
Extra Columns	no*	no	yes	yes	yes*	no	no
EAV	yes	yes	yes*	yes	yes*	no*	no
CTI	no*	no	yes	yes	yes	yes	yes
LOB	yes	yes	no	no	no	no	no
Inverted Index	yes	yes	yes	yes	yes	yes	yes
OSC	yes	no	yes	yes	yes	yes	yes
NoSQL	yes	yes	yes	yes	yes	no*	no

* conditions or exceptions apply.

SQL Modes

Percona Training

http://www.percona.com/training

Table of Contents

SQL Mode

WHAT IS THE SERVER SQL MODE?

Default MySQL behaviour

- For historical reasons, MySQL's default configuration is too lax about data constraints:

```
mysql> CREATE TABLE `test` (
   `id` int(11) NOT NULL,
   `type` enum('movie','album','videogame') NOT NULL,
   `date` datetime NOT NULL,
   PRIMARY KEY (`id`)
) ENGINE=InnoDB;

mysql> INSERT INTO test (type, date) VALUES ('tv show', -1);
...
```

What happens?

Default MySQL behaviour (cont.)

```
Query OK, 1 row affected, 3 warnings (0.00 sec)

mysql> SHOW WARNINGS;
+---------+------+---------------------------------------------+
| Level   | Code | Message                                     |
+---------+------+---------------------------------------------+
| Warning | 1364 | Field 'id' doesn't have a default value     |
| Warning | 1265 | Data truncated for column 'type' at row 1   |
| Warning | 1264 | Out of range value for column 'date' at row 1 |
+---------+------+---------------------------------------------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM test;
+----+------+---------------------+
| id | type | date                |
+----+------+---------------------+
|  0 |      | 0000-00-00 00:00:00 |
+----+------+---------------------+
1 row in set (0.00 sec)
```

Why Does It Get Inserted?

- No value/NULL for primary key gets converted into integer 0 (without an auto_increment)
- Item not part of the enum gets converted into the 0th element of the enum struct: '' (invalid value)
- Invalid date gets converted into the 'zero date': 0000-00-00 00:00:00
- In a non-strict mode, those conversions only produce <u>warnings, not errors</u>

How to Fix It?

- Go “traditional”:

```
mysql> SET sql_mode = 'TRADITIONAL';

mysql> INSERT INTO test (type, date) VALUES ('tv show', -1);
ERROR 1364 (HY000): Field 'id' doesn't have a default value

mysql> INSERT INTO test VALUES (1, 'tv show', -1);
ERROR 1265 (01000): Data truncated for column 'type' at row 1

mysql> INSERT INTO test VALUES (1, 'movie', -1);
ERROR 1292 (22007): Incorrect datetime value: '-1' for column
'date' at row 1
```

SQL Modes

- sql_mode is a global and session variable that controls the behavior of MySQL (constraints and syntax)
- It holds comma-separated flags that can enabled/ certain features
- It was created for compatibility with older versions of MySQL and other DBMS

Default sql_mode

- In versions 5.5 and lower the default value for sql_mode was '' (empty – non-strict by default)
- From 5.6.6, the hardcoded default is: `NO_ENGINE_SUBSTITUTION`
  - The suggested default in Oracle's provided my.cnf is `NO_ENGINE_SUBSTITUTION,STRICT_TRANS_TABLES`
- In both cases, you probably want to change it to a stricter mode

Combination Modes

- For convenience, some sql_mode values are equivalent to a combination of several others
- For example, `SET sql_mode = 'ORACLE'` is equivalent to:

```
SET sql_mode = 'PIPES_AS_CONCAT,
ANSI_QUOTES, IGNORE_SPACE,
NO_KEY_OPTIONS, NO_TABLE_OPTIONS,
NO_FIELD_OPTIONS, NO_AUTO_CREATE_USER'
```

SQL Mode

TRADITIONAL

Common Modes: Traditional

- `'TRADITIONAL'` sets the common behavior of most RDBMS regarding data constraints
  - It produces an error instead of a warning in most cases
- **You usually want to use this for data security**
- It is a combination mode equivalent to:
  `STRICT_TRANS_TABLES, STRICT_ALL_TABLES, NO_ZERO_IN_DATE, NO_ZERO_DATE, ERROR_FOR_DIVISION_BY_ZERO, NO_AUTO_CREATE_USER, NO_ENGINE_SUBSTITUTION`

Zero Dates

- `'NO_ZERO_DATE'` disallows the `'0000-00-00'` value
- `'NO_ZERO_IN_DATE'` disallow zero months or days:

```
mysql> INSERT INTO strict values ('2014-01-00
00:00:00');
```

`sql_mode = ''` inserts the values as is, with a warning
`sql_mode = 'NO_ZERO_IN_DATE'` inserts the value '0000-00-00 00:00:00'
`sql_mode = 'NO_ZERO_IN_DATE,NO_ZERO_DATE'` returns an error

Division by Zero

- `sql_mode = 'ERROR_FOR_DIVISION_BY_ZERO'`
- Returns an error instead of the value NULL on division by (or mod) zero

Auto-User Creation

- By default, GRANT statement creates automatically a user if it did not exist
  - This can be a security issue, specially if no password is provided

```
mysql> CREATE USER 'user'@'192.168.0.100' IDENTIFIED BY 'pass';
Query OK, 0 rows affected (0.00 sec)
mysql> GRANT ALL PRIVILEGES ON *.* TO 'user'@'192.168.0.110‘;
Query OK, 0 rows affected (0.00 sec)
```

A new user has been created with different credentials and no password by mistake!

Auto-User Creation (cont.)

- SQL mode `NO_AUTO_CREATE_USER` mitigates that problem by disallowing:
  - Granting privileges to unknown user accounts, or
  - Granting privileges without setting a password (or equivalent credentials)

```
mysql> SET sql_mode = 'NO_AUTO_CREATE_USER';
Query OK, 0 rows affected (0.00 sec)
mysql> GRANT ALL PRIVILEGES ON *.* TO 'user'@'192.168.0.111';
ERROR 1133 (42000): Can't find any matching row in the user table
```

SQL Mode

ANSI

Common Modes: ANSI

- `'ANSI'` mode changes MySQL syntax to be more similar to the SQL standard
- It is a compound mode, equivalent* to:

```
SET sql_mode = 'REAL_AS_FLOAT,
PIPES_AS_CONCAT, ANSI_QUOTES,
IGNORE_SPACE';
```

*It also changes a particular behavior with subselects, see
http://dev.mysql.com/doc/refman/5.6/en/server-sql-mode.html#sqlmode_ansi

Common Modes: ANSI (cont.)

- `REAL_AS_FLOAT`
  - `real` datatype is single precision instead of double precision
- `PIPES_AS_CONCAT`
  - The `||` symbol is an alias for `concat()` instead of the logical `OR`

Common Modes: ANSI (cont.)

- `ANSI_QUOTES`
  - The double quote (") symbol is an identifier delimiter instead of a string delimiter
- `IGNORE_SPACE`
  - Allows a space between the name of a built-in function and its parameter definition: the '(' symbol
  - As a consequence, all built-in function names become reserved words

ansi Server Option

- Starting MySQL with the `--ansi` command line parameter (or setting the `ansi` configuration option under the `[mysqld]` section) changes both the SQL Mode **and the default isolation level**
- It is equivalent to:

```
--transaction-isolation=SERIALIZABLE
--sql-mode=ANSI
```

SQL Mode

ISSUES AND PERFORMANCE

SQL Modes Problems

- Changing the server SQL mode once it is running, or changing it for some session may be problematic in some cases
- Partitioning
  - If the SQL Mode changes the key function after the insertion of values, some of them may not be retrieved
- Replication
  - If master and slave have different default SQL Modes, slave can become inconsistent or replication may break

Performance

- Even if application performs all required validations and data checks, you may still want to set MySQL Strict Mode
- The performance gain is not that big, compared to the overhead of other features, and it can be disabled in appropriate cases (bulk loading)

SQL Modes and Connectors

- Some MySQL connectors do not provide a(n easy) way to monitor Warnings
  - Make sure there is a way to debug them
- Connector/J 5.0+ sets STRICT_TRANS_TABLES

Alternative Syntax for Backwards Compatibility

- MySQL uses "versioned" comments to handle new features in SQL language:

```
mysql> CREATE TABLE with_partitions (id int PRIMARY KEY)
       PARTITION BY HASH(id) PARTITIONS 4;
Query OK, 0 rows affected (0.95 sec)
mysql> SHOW CREATE TABLE with_partitions\G
*********************** 1. row ***********************
       Table: with_partitions
Create Table: CREATE TABLE `with_partitions` (
  `id` int(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50100 PARTITION BY HASH (id)
PARTITIONS 4 */
1 row in set (0.00 sec)
```

MySQL >= 5.1.0 will execute the comment; MySQL < 5.1.0 will not.

SQL Mode

INNODB STRICT MODE

InnoDB Strict Mode

- It is the equivalent of the SQL strict mode for InnoDB operations
- If it OFF (default), a CREATE TABLE or an ALTER TABLE will ignore incorrect key_block or row_format options. The action will be performed with a warning.
- If it is ON, the statements will throw an error immediately.

InnoDB Strict Mode (cont.)

```
mysql> SET GLOBAL innodb_file_per_table = 0;
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE TABLE compact (id int PRIMARY KEY)
       ENGINE=InnoDB row_format=dynamic;
Query OK, 0 rows affected, 2 warnings (0.18 sec)

mysql> show warnings;
+---------+------+--------------------------------------------------------+
| Level   | Code | Message                                                |
+---------+------+--------------------------------------------------------+
| Warning | 1478 | InnoDB: ROW_FORMAT=DYNAMIC requires innodb_file_per_table. |
| Warning | 1478 | InnoDB: assuming ROW_FORMAT=COMPACT.                   |
+---------+------+--------------------------------------------------------+
2 rows in set (0.00 sec)
```

InnoDB Strict Mode (cont.)

```
mysql> set innodb_strict_mode = ON;
Query OK, 0 rows affected (0.00 sec)

mysql> ALTER TABLE compact (id int PRIMARY KEY) ENGINE=InnoDB
row_format=dynamic;
ERROR 1031 (HY000): Table storage engine for 'compact' doesn't have this
option
mysql> set global innodb_file_per_table = 1;
Query OK, 0 rows affected (0.00 sec)
mysql> set global innodb_file_format ='Barracuda';
Query OK, 0 rows affected (0.00 sec)
mysql> ALTER TABLE compact ENGINE=InnoDB;
Query OK, 0 rows affected (0.46 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> ALTER TABLE compact row_format=dynamic;
Query OK, 0 rows affected (0.38 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

MySQL Partitioning

Percona Training

http://www.percona.com/training

Table of Contents

MySQL Partitioning

OVERVIEW

MySQL Partitioning

- Implementation of user-defined partitioning
- This is horizontal partitioning (different rows of a table may be assigned to different physical partitions)
- Since 5.1 (not compatible with partitions <5.1.6)

Overview

```
mysql> SHOW VARIABLES LIKE '%partition%';
+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| have_partitioning | YES   |
+-------------------+-------+
1 row in set (0.00 sec)
```

`have_partitioning` variable has been removed in MySQL 5.6.1, use SHOW PLUGINS instead

```
mysql> SHOW PLUGINS;
+------------+----------+----------------+---------+---------+
| Name       | Status   | Type           | Library | License |
+------------+----------+----------------+---------+---------+
| binlog     | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
| partition  | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
| ARCHIVE    | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
| BLACKHOLE  | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
| CSV        | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
| FEDERATED  | DISABLED | STORAGE ENGINE | NULL    | GPL     |
| MEMORY     | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
| InnoDB     | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
| MRG_MYISAM | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
| MyISAM     | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
+------------+----------+----------------+---------+---------+
10 rows in set (0.00 sec)
```

What Is Partitioning

- Logical splitting of tables
  - no need to create separate tables
  - no need to move chunks of data across files
- Transparent to user
  - this is not a MERGE table
- The user-selected rule by which the division of data is accomplished is known as partitioning function

What Partitions Can Do?

- Logical split
- Data can be split physically
- Granular partitions (subpartitioning)
- Different methods can be used (range, list, hash, key)

Why Partitioning?

- **For large tables!**
  - **“Divide & Conquer”**
- Easier Maintenance
  - Store historical data efficiently
  - Delete large chunks of data faster
- Performance improvement for queries
  - Single inserts faster
  - Single selects faster
  - Range selects faster
- Control data placement on devices

What's the best reasons to use MySQL Partitioning ?

- You have large tables
  - …or fast-growing ones
- You know you will always query using the partitioning column(s)
- You want to purge quickly historical tables
- You have indexes larger than the available memory
- You have some specific workloads that causes some bottlenecks

MySQL Partitioning

PARTITIONING TYPES

Partitioning Types

- In MySQL 5.6 the types of partitioning available are:
  - RANGE
  - LIST
  - COLUMNS (*)
  - HASH
  - KEY

(*) COLUMNS are variants on RANGE and LIST introduced in MySQL 5.5.0

RANGE Partitioning

- This type of partitioning assigns rows to partitions based on column values falling within a given range.
  - MySQL 5.5 adds an extension to this type: `RANGE COLUMNS`
- Ranges must be **contiguous**
- Ranges must **not** be **overlapping**
- Ranges are defined using `VALUES LESS THAN` operator

Example

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT NOT NULL,
    store_id INT NOT NULL
);
```

Let's use store_id for partitioning

Example (cont.)

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT NOT NULL,
    store_id INT NOT NULL
)
PARTITION BY RANGE (store_id) (
    PARTITION p0 VALUES LESS THAN (6),
    PARTITION p1 VALUES LESS THAN (11),
    PARTITION p2 VALUES LESS THAN (16),
    PARTITION p3 VALUES LESS THAN (21)
);
```

Each partition is defined in order, from lowest to highest. This is a requirement of the PARTITION BY RANGE syntax.

But What Happens If We Add a 21st store?

Example (cont.)

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
```

under this scheme... an ERROR !

```
    store_id INT NOT NULL
)
PARTITION BY RANGE (store_id) (
    PARTITION p0 VALUES LESS THAN (6),
    PARTITION p1 VALUES LESS THAN (11),
    PARTITION p2 VALUES LESS THAN (16),
    PARTITION p3 VALUES LESS THAN (21)
);
```

Example (cont.)

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT NOT NULL,
    store_id INT NOT NULL
)
PARTITION BY RANGE (store_id) (
    PARTITION p0 VALUES LESS THAN (6),
    PARTITION p1 VALUES LESS THAN (11),
    PARTITION p2 VALUES LESS THAN (16),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

Query OK.

Example (cont.)

- It is also possible to use an expression in VALUE LESS THAN clauses:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)
PARTITION BY RANGE ( YEAR(separated) ) (
    PARTITION p0 VALUES LESS THAN (1991),
    PARTITION p1 VALUES LESS THAN (1996),
    PARTITION p2 VALUES LESS THAN (2001),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

Day and Time

- MySQL partitioning is optimized for use with
  - `TO_DAYS()`
  - `YEAR()`
  - `TO_SECONDS()`
- However you can use other date and time function that return an integer or NULL
  - `WEEKDAY()`
  - `DAYOFYEAR()`
  - `MONTH()`
- `RANGE COLUMNS` (explained later) allows the usage of range partitioning on a `DATE` or `DATETIME` column

RANGE and TIMESTAMP column

- In MySQL 5.5.1+ it is also possible to use a `TIMESTAMP` column to partition by `RANGE` or `LIST`
  - `UNIX_TIMESTAMP()` function must be used.
  - Any other expression involving `TIMESTAMP` values are not permitted

Example

```
CREATE TABLE quarterly_report_status (
    report_id INT NOT NULL,
    report_status VARCHAR(20) NOT NULL,
    report_updated TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP
)
PARTITION BY RANGE ( UNIX_TIMESTAMP(report_updated) ) (
    PARTITION p0 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-01-01 00:00:00') ),
    PARTITION p1 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-04-01 00:00:00') ),
    PARTITION p2 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-07-01 00:00:00') ),
    PARTITION p3 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-10-01 00:00:00') ),
    PARTITION p4 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-01-01 00:00:00') ),
    PARTITION p5 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-04-01 00:00:00') ),
    PARTITION p6 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-07-01 00:00:00') ),
    PARTITION p7 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-10-01 00:00:00') ),
    PARTITION p8 VALUES LESS THAN ( UNIX_TIMESTAMP('2010-01-01 00:00:00') ),
    PARTITION p9 VALUES LESS THAN (MAXVALUE)
);
```

NULL Values with RANGE Partitioning

- For `RANGE`, the row is inserted into the lowest partition (*):

```
CREATE TABLE tn (
    c1 INT,
    c2 VARCHAR(20)
)
PARTITION BY RANGE(c1) (
    PARTITION p0 VALUES LESS THAN (-5),
    PARTITION p1 VALUES LESS THAN (0),
    PARTITION p2 VALUES LESS THAN (10),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

(*) In versions < 5.1.8, RANGE implicitly treated NULL values as 0 for partitioning selection.

NULL Values with RANGE Partitioning (cont.)

```
INSERT INTO tn VALUES (NULL, 'a null');
Query OK, 1 row affected (0.00 sec)

SELECT TABLE_NAME, PARTITION_NAME, TABLE_ROWS, AVG_ROW_LENGTH, DATA_LENGTH
FROM INFORMATION_SCHEMA.PARTITIONS
WHERE TABLE_SCHEMA = 'test' AND TABLE_NAME = 'tn';
+------------+----------------+------------+----------------+-------------+
| TABLE_NAME | PARTITION_NAME | TABLE_ROWS | AVG_ROW_LENGTH | DATA_LENGTH |
+------------+----------------+------------+----------------+-------------+
| tn         | p0             |          1 |             20 |          20 |
| tn         | p1             |          0 |              0 |           0 |
| tn         | p2             |          0 |              0 |           0 |
| tn         | p3             |          0 |              0 |           0 |
+------------+----------------+------------+----------------+-------------+
7 rows in set (0.01 sec)
```

PERCONA TRAINING

How Does It Look Like on the Filesystem?

- Using MyISAM

```
-rw-rw---- 1 mysql mysql 8.5K Apr  1 22:55 salaries.frm
-rw-rw---- 1 mysql mysql   48 Apr  1 22:55 salaries.par
-rw-rw---- 1 mysql mysql  15M Apr  1 22:56 salaries#P#p0.MYD
-rw-rw---- 1 mysql mysql  14M Apr  1 22:56 salaries#P#p0.MYI
-rw-rw---- 1 mysql mysql 2.9M Apr  1 22:56 salaries#P#p1.MYD
-rw-rw---- 1 mysql mysql 2.6M Apr  1 22:56 salaries#P#p1.MYI
-rw-rw---- 1 mysql mysql 3.2M Apr  1 22:56 salaries#P#p2.MYD
-rw-rw---- 1 mysql mysql 2.9M Apr  1 22:56 salaries#P#p2.MYI
-rw-rw---- 1 mysql mysql 3.4M Apr  1 22:56 salaries#P#p3.MYD
-rw-rw---- 1 mysql mysql 3.0M Apr  1 22:56 salaries#P#p3.MYI
-rw-rw---- 1 mysql mysql 3.6M Apr  1 22:56 salaries#P#P4.MYD
-rw-rw---- 1 mysql mysql 3.2M Apr  1 22:56 salaries#P#P4.MYI
-rw-rw---- 1 mysql mysql 3.8M Apr  1 22:56 salaries#P#p5.MYD
-rw-rw---- 1 mysql mysql 3.4M Apr  1 22:56 salaries#P#p5.MYI
-rw-rw---- 1 mysql mysql 9.3M Apr  1 22:56 salaries#P#p6.MYD
-rw-rw---- 1 mysql mysql 8.3M Apr  1 22:56 salaries#P#p6.MYI
-rw-rw---- 1 mysql mysql    0 Apr  1 22:55 salaries#P#p7.MYD
-rw-rw---- 1 mysql mysql 1.0K Apr  1 22:55 salaries#P#p7.MYI
```

How Does It Look Like on the Filesystem?

- Using InnoDB with table space file per table (`innodb_file_per_table=1`)

```
-rw-rw---- 1 mysql mysql 8.5K Apr  1 22:51 salaries.frm
-rw-rw---- 1 mysql mysql   48 Apr  1 22:51 salaries.par
-rw-rw---- 1 mysql mysql  44M Apr  1 22:52 salaries#P#p0.ibd
-rw-rw---- 1 mysql mysql  15M Apr  1 22:52 salaries#P#p1.ibd
-rw-rw---- 1 mysql mysql  15M Apr  1 22:52 salaries#P#p2.ibd
-rw-rw---- 1 mysql mysql  16M Apr  1 22:52 salaries#P#p3.ibd
-rw-rw---- 1 mysql mysql  16M Apr  1 22:52 salaries#P#P4.ibd
-rw-rw---- 1 mysql mysql  17M Apr  1 22:52 salaries#P#p5.ibd
-rw-rw---- 1 mysql mysql  30M Apr  1 22:53 salaries#P#p6.ibd
-rw-rw---- 1 mysql mysql  96K Apr  1 22:51 salaries#P#p7.ibd
```

RANGE Partitioning Exercise

- Create four partitions (id) for the table title with approximately the same number of records
- Verify the partitioning by counting the number of records on each one by using the INFORMATION_SCHEMA tables

LIST Partitioning

- Each partition must be explicitly defined
- Each partition is defined and selected based on the membership of a column value in a set of value lists.
- Defined using `PARTITION BY LIST(expr)` where expr is a column value or an expression based on a column value returning an integer
- Partitions are defined using `VALUES IN (value_list)` where value_list is a comma-separated list of integers

Example

- Suppose that there are 20 video stores distributed among 4 franchises as shown in the following table.

Region	Store IDs
North	3, 5, 6, 9, 17
East	1, 2, 10, 11, 19, 20
West	4, 12, 13, 14, 18
Central	7, 8, 15, 16

Example (cont.)

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)
PARTITION BY LIST(store_id) (
    PARTITION pNorth VALUES IN (3,5,6,9,17),
    PARTITION pEast VALUES IN (1,2,10,11,19,20),
    PARTITION pWest VALUES IN (4,12,13,14,18),
    PARTITION pCentral VALUES IN (7,8,15,16)
);
```

LIST partitions do not need to be declared in any particular order

PERCONA TRAINING

LIST Partitioning (cont.)

- Unlike the case with `RANGE` partitioning, there is no "catch-all" such as `MAXVALUE`; all expected values for the partitioning expression should be covered in

  `PARTITION ... VALUES IN (...)` clauses.

NULL Values with LIST Partitioning

- Unlike RANGE, there is no implicit usage of NULL values with LIST partitioning
- NULL is only allowed as a partitioning expression value if it one of the explicit members of a list:

```
CREATE TABLE tn (
    c1 INT,
    c2 VARCHAR(20)
)
PARTITION BY LIST(c1) (
    PARTITION p0 VALUES IN (0, 3, 6),
    PARTITION p1 VALUES IN (1, 4, 7),
    PARTITION p2 VALUES IN (2, 5, 8)
);
```

NULL Values with LIST Partitioning (cont.)

```
INSERT INTO tn VALUES (NULL, 'a null');

CREATE TABLE tn2 (
    c1 INT,
    c2 VARCHAR(20)
)
PARTITION BY LIST(c1) (
    PARTITION p0 VALUES IN (0, 3, 6, NULL),
    PARTITION p1 VALUES IN (1, 4, 7),
    PARTITION p2 VALUES IN (2, 5, 8)
);

INSERT INTO tn2 VALUES (NULL, 'a null');
```

ERROR 1504 (HY000)

Query OK.

COLUMNS Partitioning

- Introduced in MySQL 5.5.0 `COLUMNS` partitioning enables the use of multiple columns in partitioning keys.
- Available for `RANGE` and `LIST`
- Data types supported:
  - All integer types: TINYINT, SMALLINT, MEDIUMINT, INT and BIGINT
  - DATE and DATETIME
  - String types: CHAR, VARCHAR, BINARY and VARBINARY
- No other numeric data types supported like DECIMAL or FLOAT
- TEXT and BLOB are not supported

RANGE COLUMNS

- RANGE COLUMNS differs from RANGE in the following ways:
  - It does not accept expressions, only names of columns
  - It accepts a list of one or more columns
  - It is not restricted to integer columns

```
CREATE TABLE rcx (
    a INT,
    b INT,
    c CHAR(3),
    d INT
)
PARTITION BY RANGE COLUMNS(a,d,c) (
    PARTITION p0 VALUES LESS THAN (5,10,'ggg'),
    PARTITION p1 VALUES LESS THAN (10,20,'mmmm'),
    PARTITION p2 VALUES LESS THAN (15,30,'sss'),
    PARTITION p3 VALUES LESS THAN (MAXVALUE,MAXVALUE,MAXVALUE)
);
```

RANGE COLUMNS (cont.)

```
CREATE TABLE members (
    firstname VARCHAR(25) NOT NULL,
    lastname VARCHAR(25) NOT NULL,
    username VARCHAR(16) NOT NULL,
    email VARCHAR(35),
    joined DATE NOT NULL
)
PARTITION BY RANGE COLUMNS(joined) (
    PARTITION p0 VALUES LESS THAN ('1960-01-01'),
    PARTITION p1 VALUES LESS THAN ('1970-01-01'),
    PARTITION p2 VALUES LESS THAN ('1980-01-01'),
    PARTITION p3 VALUES LESS THAN ('1990-01-01'),
    PARTITION p4 VALUES LESS THAN MAXVALUE
);
```

LIST COLUMNS

- Same features and restrictions as RANGE COLUMNS

```
CREATE TABLE customers_1 (
    first_name VARCHAR(25),
    last_name VARCHAR(25),
    street_1 VARCHAR(30),
    street_2 VARCHAR(30),
    city VARCHAR(15),
    renewal DATE
)
PARTITION BY LIST COLUMNS(city) (
    PARTITION pRegion_1 VALUES IN('Oskarshamn', 'Högsby', Mönsterås'),
    PARTITION pRegion_2 VALUES IN('Vimmerby', 'Hultsfred', Västervik'),
    PARTITION pRegion_3 VALUES IN('Nässjö', 'Eksjö', 'Vetlanda'),
    PARTITION pRegion_4 VALUES IN('Uppvidinge', 'Alvesta', 'Växjo')
);
```

HASH Partitioning

- Partitions are selected based on the value returned by a user-defined expression that operates on column values.
- The function is a valid expression that yields a nonnegative integer value.
- Used to ensure an even distribution of data
- Defined by `PARTITION BY HASH (expr)` and followed by with `PARTITIONS num`
- An extension to this type, `LINEAR HASH`, is also available.

Example

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)
PARTITION BY HASH(store_id)
PARTITIONS 4;
```

The expression must return a non-constant, non-random integer value and must not contain any prohibited constructs (see restrictions and limitations topic).

Example

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)
PARTITION BY HASH(store_id)
PARTITIONS 4;
```

If omitted, the default number of partitions is 1.

HASH modulus

- To determine which partition needs to be used, MySQL performs a modulus of the user function:

```
N = MOD(expr, num)
```

(N is the partition number)

```
CREATE TABLE t1 (col1 INT, col2 CHAR(5), col3 DATE)
    PARTITION BY HASH( YEAR(col3) )
    PARTITIONS 4;
```

- If you insert a record into `t1` whose `col3` value is `'2012-04-02'`, in which partition will it be stored ?

```
mysql> select MOD(YEAR('2012-04-02'), 4);
+----------------------------+
| MOD(YEAR('2012-04-02'),4) |
+----------------------------+
|                          0 |
+----------------------------+
```

Hash Partitioning Exercise

- Revert the title table to the previous status -no partitions
- Create 4 partitions BY HASH based on the primary key
- Which partition has row #3? And #4?

KEY Partitioning

- Similar to `HASH` except that the user-defined expression is supplied transparently by MySQL
  - Same algorithm as `PASSWORD()` is used
  - `MD5()` in MySQL Cluster
- Defined using `PARTITION BY KEY`
- Same as `HASH` but
  - `KEY` is used rather than `HASH`
  - `KEY` takes only a list of column names (0 or more)
  - Columns are not restricted to integer or `NULL`

Example

```
CREATE TABLE k1 (
     id INT NOT NULL,
     name VARCHAR(20)
)
PARTITION BY KEY(name)
PARTITIONS 2;

CREATE TABLE k1 (
     id INT NOT NULL PRIMARY KEY,
     name VARCHAR(20)
)
PARTITION BY KEY()
PARTITIONS 2;
```

No column specified, PRIMARY KEY is used

```
CREATE TABLE k1 (
     id INT NOT NULL,
     name VARCHAR(20),
UNIQUE KEY (id)
)
PARTITION BY KEY()
PARTITIONS 2;
```

No PRIMARY KEY, but there is a UNIQUE KEY that is used for the partitioning key.
If the unique key was defined as NULL, the statement would fail

NULL values with HASH and KEY Partitioning

- In the case of using HASH or KEY partitioning, the expression returning a NULL value is treated as if it returned a zero:

```
CREATE TABLE tn (
    c1 INT,
    c2 VARCHAR(20)
)
PARTITION BY HASH(c1)
PARTITIONS 2;
```

NULL values with HASH and KEY Partitioning (cont.)

```
INSERT INTO tn VALUES (NULL, 'a null');
INSERT INTO tn VALUES (0, 'a zero');

SELECT TABLE_NAME, PARTITION_NAME, TABLE_ROWS, AVG_ROW_LENGTH, DATA_LENGTH
FROM INFORMATION_SCHEMA.PARTITIONS
WHERE TABLE_SCHEMA = 'p' AND TABLE_NAME ='tn';
+------------+----------------+------------+----------------+-------------+
| TABLE_NAME | PARTITION_NAME | TABLE_ROWS | AVG_ROW_LENGTH | DATA_LENGTH |
+------------+----------------+------------+----------------+-------------+
| tn         | p0             |          2 |             20 |          20 |
| tn         | p1             |          0 |              0 |           0 |
+------------+----------------+------------+----------------+-------------+
2 rows in set (0.00 sec)
```

MySQL Partitioning

PARTITION MANAGEMENT

Partition Management

- It's possible to add, drop, redefine, merge, or split existing partitions
- All these actions are done via `ALTER TABLE`
  - Most of these cannot be combined with other operations on the same `ALTER TABLE`
- We make 2 distinctions in management of partitions:
  - `RANGE` and `LIST`
  - `HASH` and `KEY`

Management of LIST and RANGE Partitions

- Dropping a partition **with all its data**

```
ALTER TABLE table1
DROP PARTITION p2;
```

With `LIST` you can no longer insert into the table any rows that should have been added in the deleted partition

- Remove all data but keep the partitioning scheme

```
TRUNCATE TABLE table1;
```

- Add an empty partition

```
ALTER TABLE table1 ADD PARTITION
(PARTITION p3 VALUES LESS THAN (n));
```

Management of LIST and RANGE Partitions (cont.)

- Reorganize table partitions **without loosing data**

```
ALTER TABLE table1 REORGANIZE PARTITION
partitions_list INTO (partition_definitions);
```

  - For tables partitioned by `RANGE`, you can reorganize only adjacent partitions; you cannot skip over range partitions.
  - You cannot use `REORGANIZE PARTITION` to change the table's partitioning type, the partitioning expression or column. Use `ALTER TABLE ... PARTITION BY ...` or `DROP` and recreate the table instead.

Management of HASH and KEY Partitions

- You cannot drop partitions from tables partitioned with these types in the same way we did it with `RANGE` or `LIST`.
- You can merge `HASH` or `KEY` partitions using `ALTER TABLE ... COALESCE PARTITION` statement.
  - It requires an argument with the number of partitions to drop/merge
- You can increase the number of partitions using `ALTER TABLE ... ADD PARTITION PARTITIONS` statement.

Maintenance of Partitions

- There are several `ALTER TABLE` extensions that allow maintenance operations on a partition
  - They also accept a list of partitions or the alias `ALL`
- `ALTER TABLE ... REBUILD PARTITION`
  - For defragmenting, is equivalent to dropping all records and reinserting them
- `ALTER TABLE ... OPTIMIZE PARTITION`
  - Equivalent to `CHECK PARTITION`, `ANALYZE PARTITION`, and `REPAIR PARTITION`

Maintenance of Partitions (cont.)

- `ALTER TABLE ... ANALYZE PARTITION`
  – reads and stores the key distributions for partitions.
- `ALTER TABLE ... REPAIR PARTITION`
  – attempts to repair corrupted partitions.
- `ALTER TABLE ... CHECK PARTITION`
  – checks partitions for errors

TRUNCATE and REMOVE Partitions

- You can also truncate partitions using:

  `ALTER TABLE ... TRUNCATE PARTITION ...`

  - You call use `ALTER TABLE ... TRUNCATE PARTITION ALL` to truncate all partitions in the table.

- For removing all partition definitions from a table, you can use:

  `ALTER TABLE ... REMOVE PARTITIONING;`

EXCHANGE PARTITION

- In MySQL 5.6 you can exchange single partition or subpartition data with a single unpartitioned table in a fast way with `EXCHANGE PARTITION`:

```
CREATE TABLE t1 (
    col1 int DEFAULT NULL,
    col2 char(5) DEFAULT NULL
) PARTITION BY RANGE (col1)(
    PARTITION p0 VALUES LESS THAN (0),
    PARTITION p1 VALUES LESS THAN (10),
    PARTITION p2 VALUES LESS THAN MAXVALUE
);
```

EXCHANGE PARTITION (cont.)

```
CREATE TABLE t2 like t1;

ALTER TABLE t2 REMOVE PARTITIONING;

INSERT INTO t1 VALUES (-1, 'a'), (0, 'b'), (2, 'd'),(4,
'a'), (9, 'b'), (10, 'd'),(15, 'a');

ALTER TABLE t1 EXCHANGE PARTITION p1 WITH TABLE t2;

SELECT * FROM t2;
+------+------+
| col1 | col2 |
+------+------+
|    0 | b    |
|    2 | d    |
|    4 | a    |
|    9 | b    |
+------+------+
```

PERCONA TRAINING

Obtaining Information About Partitions

- You can use `SHOW CREATE TABLE`
- You can `SHOW TABLE STATUS` to determine whether a table is partitioned
- Using the statement `EXPLAIN PARTITIONS SELECT` to see which partitions are being used
- Query the `INFORMATION_SCHEMA.PARTITIONS` table

```
SELECT TABLE_NAME, PARTITION_NAME, TABLE_ROWS,
       AVG_ROW_LENGTH, DATA_LENGTH
FROM INFORMATION_SCHEMA.PARTITIONS
WHERE TABLE_SCHEMA = 'p' AND TABLE_NAME LIKE 'table1';
```

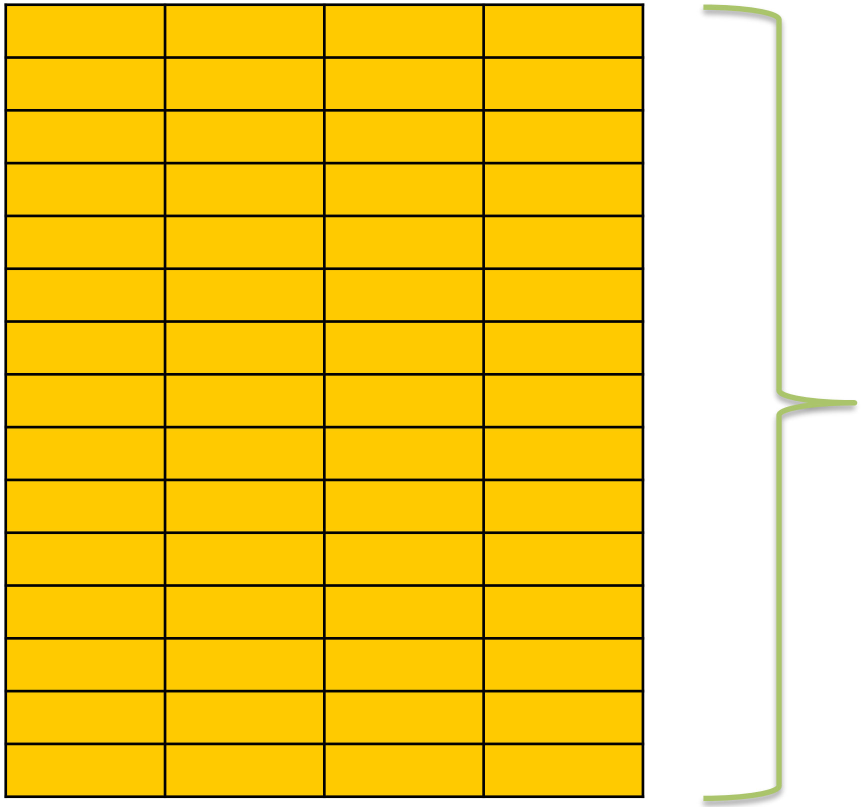
MySQL Partitioning

PARTITION PRUNING

Partition Pruning

- *“Do not scan partitions where there can be no matching values”*
- The query optimizer can perform pruning whenever a `WHERE` condition can be reduced to:
  - `partition_column = constant`
  - `partition_column IN (constant1, constant2, ..., constantN)`
- In 5.1 a query against a table partitioned by `KEY` and having a composite partitioning key could pruned only if all columns of that partitioning key were compared in the `WHERE` clause. This is not more the case in 5.5+

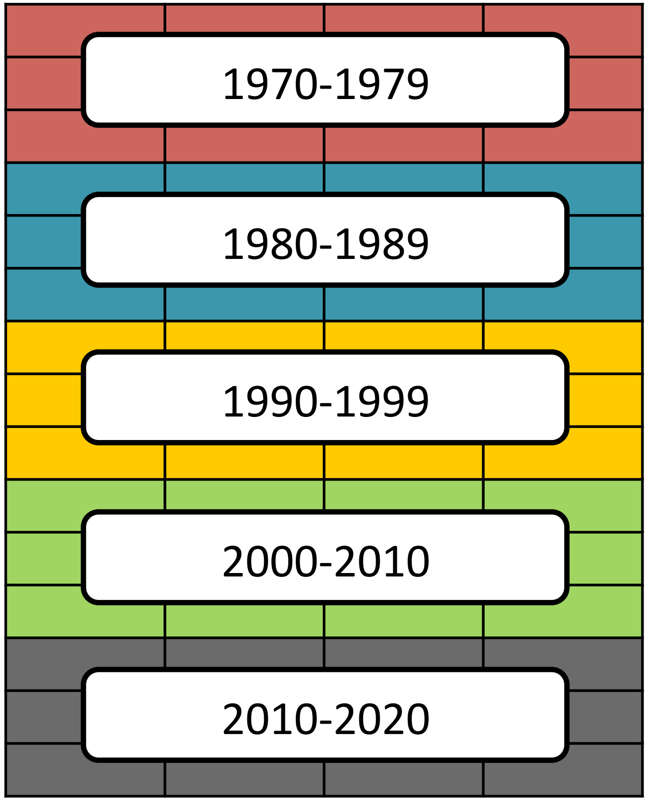
Example



- Retrieve a single record from an unpartitioned table.

SELECT * FROM table1
WHERE col_x = 1996;

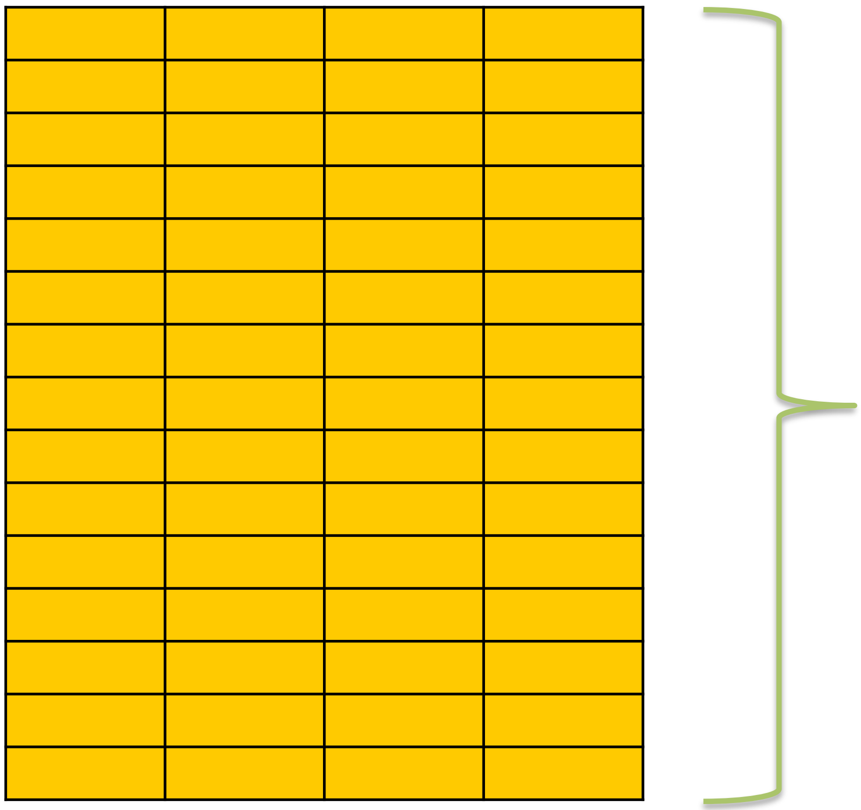
Example (cont.)



- Retrieve a single record from a partitioned table.

```
SELECT * FROM table1
WHERE col_x = 1996;
```

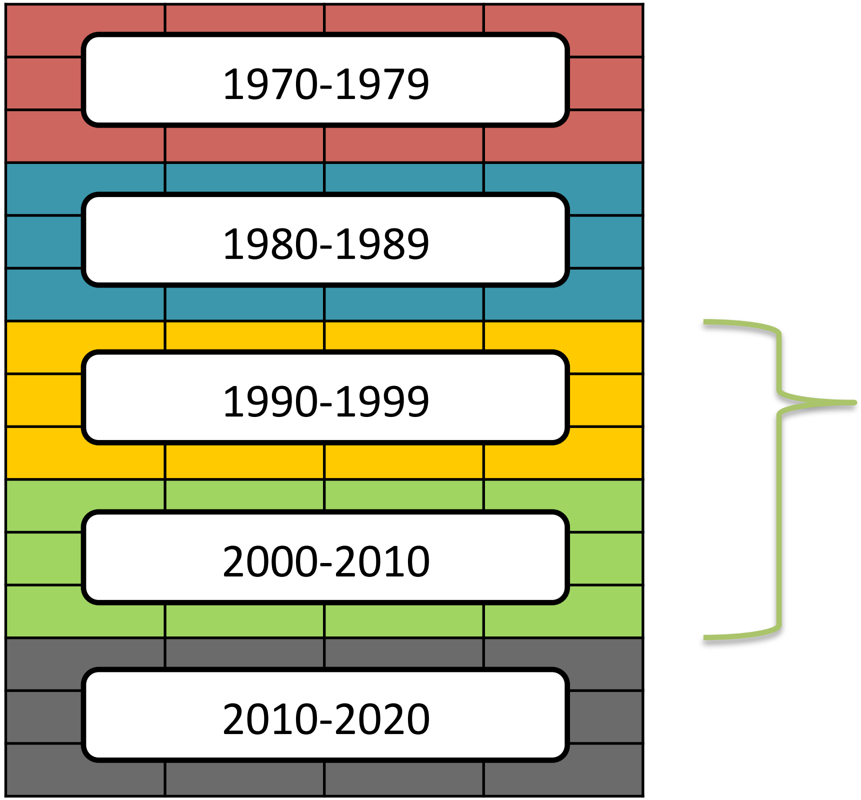
Example (cont.)



- Retrieve a range from an unpartitioned table.

```
SELECT * FROM table1
WHERE col_x BETWEEN 1997
and 2002;
```

Example (cont.)



- Retrieve a range from a partitioned table.

```
SELECT * FROM table1
WHERE col_x BETWEEN 1997
and 2002;
```

EXPLAIN PARTITIONS

- In order to check the effectiveness of partition pruning on our query plans, we can use `EXPLAIN PARTITIONS`:

```
CREATE TABLE employees (
  `id` int(11) NOT NULL,
  `fname` varchar(30) DEFAULT NULL,
  `lname` varchar(30) DEFAULT NULL,
  `hired` date NOT NULL DEFAULT '1970-01-01',
  `separated` date NOT NULL DEFAULT '9999-12-31',
  `job_code` int(11) DEFAULT NULL,
  `store_id` int(11) DEFAULT NULL)
PARTITION BY RANGE ( YEAR(separated))(
     PARTITION p0 VALUES LESS THAN (1991) ENGINE = InnoDB,
     PARTITION p1 VALUES LESS THAN (1996) ENGINE = InnoDB,
     PARTITION p2 VALUES LESS THAN (2001) ENGINE = InnoDB,
     PARTITION p3 VALUES LESS THAN MAXVALUE ENGINE = InnoDB
);
```

EXPLAIN PARTITIONS (cont.)

```
mysql> EXPLAIN PARTITIONS
    -> SELECT * FROM employees
    -> WHERE separated >= '1997-1-1' and separated <= '2003-12-31';
+----+-------------+-----------+------------+------+---------------+------+---------+------+------+-------------+
| id | select_type | table     | partitions | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+----+-------------+-----------+------------+------+---------------+------+---------+------+------+-------------+
|  1 | SIMPLE      | employees | p2,p3      | ALL  | NULL          | NULL | NULL    | NULL |    2 | Using where |
+----+-------------+-----------+------------+------+---------------+------+---------+------+------+-------------+
1 row in set (0.00 sec)
```

Partition Lock Pruning

- In MySQL 5.6.6+, storage engines that only have table-level locking, like MyISAM will only lock the affected partitions
- Only the partitions used, according to the `WHERE` clauses, will be locked for `SELECT` queries and DML statements
- Also for `ALTER TABLE ... TRUNCATE PARTITION` and `ALTER TABLE ... EXCHANGE PARTITION`
- Engines like InnoDB are not impacted by lock pruning

Partition Selection (5.6)

- In MySQL 5.6 it is possible to execute `SELECT`s and `DML`s on one or several specific partitions:

```
SELECT * FROM employees PARTITION (p0, p1);
DELETE from employees PARTITION (p3) where separated >=
'2013-1-1';
```

- It will not warn(*) about impossible conditions unless the action fails:

```
DELETE from employees PARTITION (p1)
where separated >= '2013-1-1';
INSERT INTO employees PARTITION (p1) (id, separated)
values (1000, '2013-1-1');
```

Query OK.

ERROR 1748 (HY000)

(*) "`No matching rows after partition pruning`" will be show on EXPLAIN

MySQL Partitioning

RESTRICTIONS AND LIMITATIONS

Partitioning Keys, Primary Keys and Unique Keys

- The **RULE** :

  All columns used in the partitioning expression for a partitioned table must be part of every unique key that the table may have.

- In other words:
  – Every unique key on the table must use every column in the table's partitioning expression.

Partitioning Keys, Primary Keys and Unique Keys (cont.)

```
mysql> create table t3 (
            col1 INT NOT NULL,
            col2 DATE NOT NULL,
            col3 INT NOT NULL,
            col4 INT NOT NULL,
            UNIQUE KEY (col1, col2, col3),
            UNIQUE KEY(col1)
       ) PARTITION BY HASH(col3) PARTITIONS 4;
```

ERROR 1503 (HY000): A UNIQUE INDEX must include all columns in the table's partitioning function

Partitioning Keys, Primary Keys and Unique Keys (cont.)

```
mysql> create table t3 (
           col1 INT NOT NULL,
           col2 DATE NOT NULL,
           col3 INT NOT NULL,
           col4 INT NOT NULL,
           UNIQUE KEY (col1, col2, col3),
           UNIQUE KEY(col1)
       ) PARTITION BY HASH(col1) PARTITIONS 4;
```

Query OK, 0 rows affected (0.05 sec)

Incompatible Partitioning Keys

- Prohibited constructs:
  - stored procedures, stored functions, UDF's, plugins
  - declared variables or user variables
  - Subqueries, even if they return integer values
- Arithmetic and logical operators
  - **+**, **-** and ***** are permitted in partitioning expressions but the result must be an integer or **NULL**(*)
  - **DIV** operator is also supported but not **/**
  - **|**, **&**, **^**, **<<**, **>>** and **~** are not permitted

(*) Except in the case of `[LINEAR] KEY`

Server SQL mode

- The SQL mode in effect at creation time is not preserved.
- A change in the SQL mode could lead to corruption or loss of data as the results of many MySQL functions and operators may change.
- It is strongly recommended that you never change the server SQL mode after creating partitioned tables !

Performance Considerations

- File system operations affect partitioning and repartitioning operations
  - `large_files_support =1`
  - Tune `open_files_limit`
  - `myisam_max_sort_file_size` may improve performance for MyISAM tables
  - Use `innodb_file_per_table` for InnoDB tables.
- Table locks
  - Partitioning operation takes a write lock on the table
- Storage engine
  - Partitioning operations, queries and update are faster on MyISAM

Performance Considerations (5.5+)

- Performance with `LOAD DATA`
  - From MySQL 5.5, it uses buffering to improve performance.
  - 130KB of memory per partition are used by the buffer
- Per-partition key caches
  - MySQL 5.5+ support key caches for partitioned MyISAM tables.
  - `CACHE INDEX` and `LOAD INDEX INTO CACHE`

Different Storage Engines for Partitions

- Despite having the syntax for having per-partition definition of engines, it cannot be used, and a single engine must be used at table level:

```
CREATE TABLE t1 (
    col1 int DEFAULT NULL,
    col2 char(5) DEFAULT NULL
) PARTITION BY RANGE (col1) (
    PARTITION p0 VALUES LESS THAN (0) ENGINE = MyISAM,
    PARTITION p1 VALUES LESS THAN (10) ENGINE = InnoDB,
    PARTITION p2 VALUES LESS THAN MAXVALUE ENGINE = ARCHIVE
);
```

ERROR 1497 (HY000): The mix of handlers in the partitions is not allowed in this version of MySQL

Other Restrictions

- A table can have a maximum of 8192 partitions (1024 for versions prior to 5.6.7)
- Foreign keys are not supported
- FULLTEXT indexes are not supported
- Spatial types (POINT, GEOMETRY, ...) are not supported
- Temporary tables cannot be partitioned
- Log tables cannot be partitioned
- DELAYED option not supported
- DATA DIRECTORY and INDEX DIRECTORY are ignored in table-level
- Query Cache cannot be used for partitioned tables (5.5.23+, bugged before, see #53775)

"my movies" Exercise

Percona Training

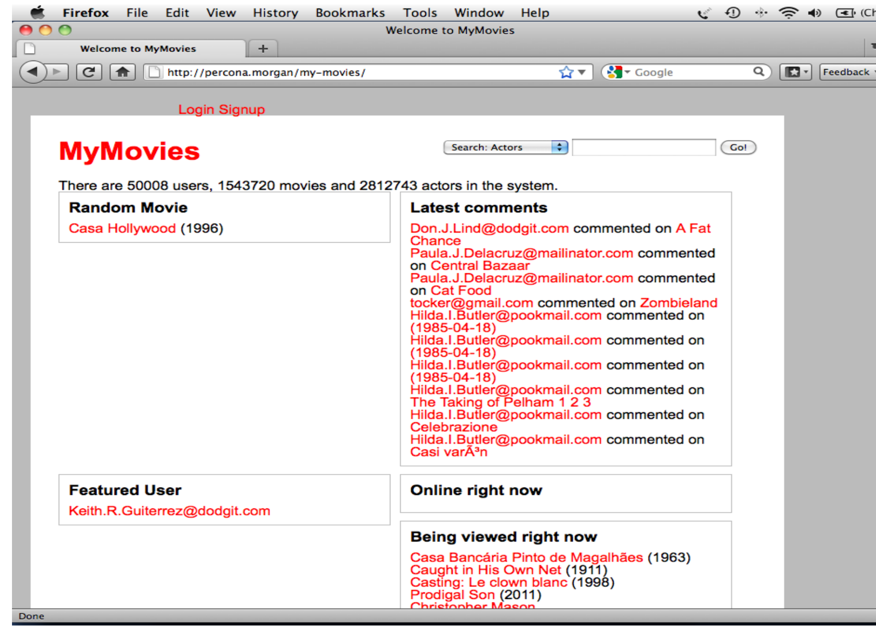
http://www.percona.com/training

Your Tasks

- Setup pt-query-digest to find slow queries.
- Use instrumentation (New Relic, Intrumentation for PHP) and your own intuition to add needed indexes and remove common offending design patterns.

Sample Application

- **MyMovies**
  - IMDB clone application.
  - Using sample data that is freely available for download.



Main Functions

- Home Page
- View a Movie
- View an Actor
- View a User
- Signup or Login
- Write comments
- Rate a movie
- Add Favourites
- Add Friends
- Search

Please check that you can complete all of these tasks.

These are the main functions that you will need to optimize.

The Focus of This Exercise

- Optimize the application **as much as you can**.
  - You will be making database changes.
  - You will be making code changes.

Your Goal

- When it comes to optimization time, you are encouraged to cheat your way out.
- For example, "there are 2812743 actors in the system," but you can calculate it differently:
  - `SELECT COUNT(*) FROM actors;`
  - May change to:
  - `SELECT MAX(id) FROM actors;`

> Please seek approval before making these changes.
> Your instructor may say no if functionality changes too much.

The Instructor's Role

- **Number #1 Goal:**
  - We want you to teach you to be methodical.
  - We want you to use data to support your decisions.

The Instructor's Role (cont.)

- **Number #2 Goal:**
  - We want to be a sounding board for your strategies.
  - Not sure what the risks are of a particular change? Ask.
  - Want to know how much change you should expect from reducing the number of queries? Ask.

You can work in teams!

- If you are not in teams, please form teams now.
- We suggest that each team should have someone who knows at least a bit of PHP.
  - But you need not be an expert.

The Rules

- Don't return bogus results.
- Don't delete data.
- Ask before degrading/changing functionality.
- Ask for help setting up new instances.
- **No caching.**
  – Except in MySQL buffers or tables.

How Do We Generate Load?

- Generate it yourself to start with.
- Please ask me when you want me to stress the application with rapid traffic.

Response Time Goals

Page	Response Time	Notes
Home page	200ms	Must load as fast as possible. This is the entry point for visitors, and has the most front-end cache misses.
Search page	1000ms	Users are more tolerant if search requests take longer.
All other pages	500ms	

Already Finished?

- Extra credit exercises:
  - Setup replication and implement application-level basic read/write splitting.
  - Implement a high level (Varnish/Squid) and/or a low level (Memcached) cache to the application.
  - Implement a hash index, FULLTEXT index or a soundex index on the table `title`.
  - Implement partitioning or sharding on the table `title`.

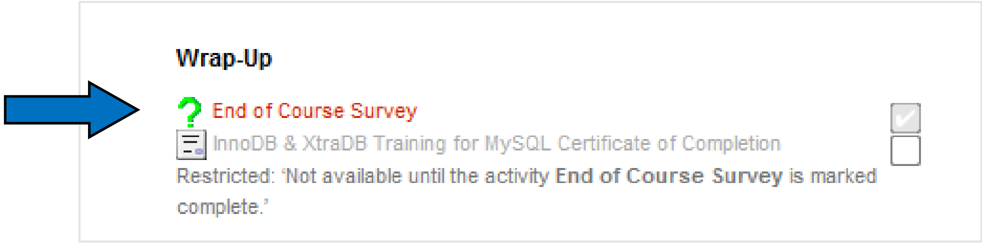
Conclusion

Percona Training

http://www.percona.com/training

Feedback Survey

- You will find it as the last link at the bottom of the Moodle page for this course.
- After completing it, you will have access to your certificate of completion for this course.



- The survey is anonymous, but we encourage you to leave your name and email, so we can follow up.

Thank you!

- The Percona Training Homepage
  http://www.percona.com/training/